

Hands On DarkBASIC Pro

Volume 2

A Self-Study Guide to Games Programming

Alistair Stewart

This is the second volume of an easy-to-follow, self-study introduction to games software development using DarkBASIC Pro - a programming language designed specifically for creating video games on a PC.

The majority of the book covers 3D programming. It begins with basic 3D concepts and goes on to cover creating 3D shapes, texturing, cameras, lighting, collision detection, shaders, importing 3D models, limbs, 3D animation, creating terrains, particles, BSPs, vertex manipulation, 3D mathematics, and physics engines (ODE).

Other topics include creating multi-user games, FTP file transfers, how to add new statements to DarkBASIC Pro using DLLs, and memory manipulation.

Everything is explained simply in a step-by-step style.

The numerous examples and exercises (with solutions included) lead you through both the basics and complexities of each topic.

There are also several complete games for you to study and modify.

Files used in the examples can be downloaded from our website.

In Volume 1

The first volume contains a complete introduction to programming in general and DarkBASIC Pro in particular. Other topics covered include strings, arrays, records, file handling, bitmaps, images, music, sound, video, sprites, 2D vectors, mouse and joystick controls, as well as several complete games.

Digital Skills

www.digital-skills.co.uk



Hands On DarkBASIC Pro

Volume 2

A Self-Study Guide to Games Programming

Alistair Stewart

Digital Skills

Milton
Barr
Girvan
Ayrshire
KA26 9TY

www.digital-skills.co.uk

Copyright © Alistair Stewart 2006

All rights reserved.

No part of this work may be reproduced or used in any form without the written permission of the author.

Although every effort has been made to ensure accuracy, the author and publisher accept neither liability nor responsibility for any loss or damage arising from the information in this book.

All brand names and product names are trademarks of their respective companies and have been capitalised throughout the text.

DarkBASIC Professional is produced by The Game Creators Ltd

Printed September 2006
2nd Printing November 2006
3rd Printing July 2007
4th Printing February 2008
5th Printing September 2009 (with corrections)
6th Printing July 2010

Title : Hands On DarkBASIC Pro Volume 2

ISBN-10 : 1-874107-09-2
ISBN-13 : 978-1-874107-09-5

Other Titles Available:

Hands On DarkBASIC Pro Volume 1
Hands On Milkshape
Hands On Pascal
Hands On C++
Hands On Java
Hands On XHTML

TABLE OF CONTENTS

Chapter 30 3D Concepts and Terminology

The 3D World	744
Introduction	744
The Coordinate System	744
Axes	744
Planes	745
Points	746
World Units	747
Local Axes	747
Rotation	748
3D Vectors	748
Object Terminology	749
Textures	750
Images with an Alpha Channel	750
Cameras	751
Lights	752
Summary	753

Chapter 31 3D Primitives

3D Primitives	756
Introduction	756
Creating a Cube	756
The MAKE OBJECT CUBE Statement	756
Creating Other Primitives	757
The MAKE OBJECT BOX Statement	757
The MAKE OBJECT SPHERE Statement	758
The MAKE OBJECT CYLINDER Statement	759
The MAKE OBJECT CONE Statement	760
The MAKE OBJECT PLAIN Statement	760
The MAKE OBJECT TRIANGLE Statement	761
Positioning an Object	762
The POSITION OBJECT Statement	762
The MOVE OBJECT Statement	764
Rotating Objects - Absolute Rotation	765
The XROTATE OBJECT Statement	766
The YROTATE OBJECT Statement	767
The ZROTATE OBJECT Statement	767
The ROTATE OBJECT Statement	768
The SET OBJECT ROTATION Statement	769
Rotating Objects - Relative Rotation	769
The PITCH OBJECT Statement	770

The TURN OBJECT Statement	770
The ROLL OBJECT Statement	771
The POINT OBJECT Statement	771
The MOVE OBJECT <i>distance</i> Statement	772
The FIX OBJECT PIVOT Statement	773
Resizing Objects	775
The SCALE OBJECT Statement	775
Showing and Hiding Objects	776
The HIDE OBJECT Statement	776
The SHOW OBJECT Statement	776
The DELETE OBJECT Statement	777
The DELETE OBJECTS Statement	777
Copying a 3D Object	778
The CLONE OBJECT Statement	778
The INSTANCE OBJECT Statement	779
Retrieving Data on 3D Objects	779
The OBJECT EXIST Statement	779
The OBJECT POSITION Statement	780
The OBJECT VISIBLE Statement	780
The OBJECT SIZE Statement	781
The OBJECT ANGLE Statement	781
Controlling an Object's Rotation Using the Mouse	782
Wireframe and Culling	783
The SET OBJECT WIREFRAME Statement	783
The SET OBJECT CULL Statement	784
Storage Methods	785
The SET GLOBAL OBJECT CREATION Statement	785
Summary	786
Merging Primitives	788
Introduction	788
The Statements	788
The PERFORM CSG UNION Statement	788
The PERFORM CSG DIFFERENCE Statement	790
The PERFORM CSG INTERSECTION Statement	790
Summary	791
Solutions	792

Chapter 32

Texturing

Adding Texture	798
Introduction	798
Loading a Texture Image	798
Using the Image as a Texture	798
The TEXTURE OBJECT Statement	798
Mipmaps	799
The LOAD IMAGE Statement Again	800

Tiling	801
The SCALE OBJECT TEXTURE Statement	802
Seamless Tiling	804
Video Texture	805
The PLAY ANIMATION TO IMAGE Statement	805
Other Texture Effects	807
The SET OBJECT TEXTURE Statement	807
The SCROLL OBJECT TEXTURE Statement	808
The SET OBJECT TRANSPARENCY Statement	810
The SET DETAIL MAPPING ON Statement	811
The SET OBJECT FILTER Statement	813
Summary	814
Other Visual Effects	815
Introduction	815
Changing Colour and Transparency	815
The COLOR OBJECT Statement	815
The GHOST OBJECT ON Statement	816
The GHOST OBJECT OFF Statement	817
The FADE OBJECT Statement	817
Summary	819
Images with an Alpha Channel	820
Introduction	820
Using Images with an Alpha Channel	820
Summary	821
Creating a Complex 3D Shape	822
Introduction	822
Designing the Castle	822
Gathering the Components	823
Creating the Code	823
The Code	824
Sky Spheres	828
Summary	830
Solutions	831

Chapter 33

Cameras

Camera Basics	836
Introduction	836
Positioning the Camera	836
The POSITION CAMERA Statement	836
The MOVE CAMERA Statement	837
Changing the Viewpoint	838
The POINT CAMERA Statement	838
The ROTATE CAMERA Statement	838
The SET CAMERA ROTATION Statement	840
The XROTATE CAMERA Statement	840

The YROTATE CAMERA Statement	841
The ZROTATE CAMERA Statement	841
The PITCH CAMERA Statement	842
The TURN CAMERA Statement	842
The ROLL CAMERA Statement	843
Retrieving Camera Data	844
The CAMERA POSITION Statement	844
The CAMERA ANGLE Statement	844
Modifying Camera Characteristics	845
The SET CAMERA VIEW Statement	845
The SET CAMERA ASPECT Statement	846
The SET CAMERA FOV Statement	847
The SET CAMERA RANGE Statement	848
Summary	849
Controlling Camera Movement	851
Introduction	851
Automatic Camera Placement	851
The AUTOCAM Statement	851
Following the Action	852
The SET CAMERA TO FOLLOW Statement	853
Giving the Player Control of the Camera	856
The CONTROL CAMERA USING ARROWKEYS Statement	856
The AUTOMATIC CAMERA COLLISION Statement	858
Controlling the Camera with the Mouse	859
Summary	862
Multiple Cameras	863
Introduction	863
Using Additional Cameras	863
The MAKE CAMERA Statement	863
The COLOR BACKDROP Statement	864
The BACKDROP Statement	864
The SET CURRENT CAMERA Statement	865
The DELETE CAMERA Statement	866
Switching Between Cameras	866
Multiple Camera Output	868
The CLEAR CAMERA VIEW Statement	869
Summary	870
Advanced Camera Techniques	871
Introduction	871
The Statements	871
The SET CAMERA TO IMAGE Statement	871
The SET CAMERA TO OBJECT ORIENTATION Statement	873
The SET OBJECT TO CAMERA ORIENTATION Statement	873
The LOCK OBJECT Statement	874
The SET VECTOR3 TO CAMERA POSITION Statement	875
The SET VECTOR3 TO CAMERA ROTATION Statement	876

Summary	876
Solutions	878

Chapter 34	Lighting
-------------------	-----------------

Lighting	886
Introduction	886
Types of Lighting	886
Ambient Lighting	886
Point Lighting	886
Spot Lighting	886
Directional Lighting	886
Lighting in DarkBASIC Pro	887
The HIDE LIGHT Statement	887
The SHOW LIGHT Statement	888
The SET AMBIENT LIGHT Statement	888
The COLOR AMBIENT LIGHT Statement	889
The MAKE LIGHT Statement	889
The DELETE LIGHT Statement	890
The COLOR LIGHT Statement	890
The POSITION LIGHT Statement	891
The SET LIGHT RANGE Statement	891
The SET SPOT LIGHT Statement	892
The SET DIRECTIONAL LIGHT Statement	892
The SET POINT LIGHT Statement	893
The POINT LIGHT Statement	893
The ROTATE LIGHT Statement	895
The SET LIGHT TO OBJECT POSITION Statement	895
The SET LIGHT TO OBJECT ORIENTATION Statement	897
Retrieving Light Data	898
The LIGHT EXIST Statement	898
The LIGHT VISIBLE Statement	899
The LIGHT RANGE Statement	899
The LIGHT TYPE Statement	899
The LIGHT POSITION Statement	900
The LIGHT DIRECTION Statement	900
Fog	901
The FOG Statement	901
The FOG COLOR Statement	902
The FOG DISTANCE Statement	902
The SET OBJECT FOG Statement	903
Summary	904
Solutions	907

Meshes	912
Introduction	912
Handling Meshes	912
The MAKE MESH FROM OBJECT Statement	912
The SAVE MESH Statement	913
The LOAD MESH Statement	914
The MAKE OBJECT Statement	914
The DELETE MESH Statement	915
The MESH EXIST Statement	915
Summary	916
Limbs	917
Introduction	917
Getting Started	917
The ADD LIMB Statement	917
The MAKE OBJECT FROM LIMB Statement	919
The OFFSET LIMB Statement	920
The ROTATE LIMB Statement	920
The SCALE LIMB Statement	922
The COLOR LIMB Statement	922
The TEXTURE LIMB Statement	923
The SCALE LIMB TEXTURE Statement	925
The SCROLL LIMB TEXTURE Statement	927
The HIDE LIMB Statement	927
The SHOW LIMB Statement	928
The REMOVE LIMB Statement	928
The LINK LIMB Statement	928
The CHANGE MESH Statement	931
The GLUE OBJECT TO LIMB Statement	931
The UNGLUE OBJECT Statement	934
The SET LIMB SMOOTHING Statement	934
Creating Doors	935
Retrieving Limb Data	936
The LIMB EXIST Statement	936
The LIMB VISIBLE Statement	937
The LIMB OFFSET Statement	937
The LIMB SCALE Statement	938
The LIMB ANGLE Statement	939
The LIMB POSITION Statement	939
The LIMB DIRECTION Statement	940
The PERFORM CHECKLIST FOR OBJECT LIMBS Statement	944
The LIMB NAME\$ Statement	945
The LIMB TEXTURE Statement	946
The LIMB TEXTURE NAME Statement	946
The CHECK LIMB LINK Statement	947

Saving a Model in DBO Format	947
Introduction	947
The DBO File Format	948
Creating an Elevator Model	948
The SAVE OBJECT Statement	949
The LOAD OBJECT Statement	950
Summary	951
Solutions	953

Chapter 36 Importing 3D Objects

Importing 3D Objects	962
Introduction	962
File Formats	963
Statements for Loading and Using 3D Objects	963
The LOAD OBJECT Statement Again	963
The PLAY OBJECT Statement	965
The LOOP OBJECT Statement	966
The TOTAL OBJECT FRAMES Statement	966
Moving the Alien	967
The SET OBJECT SPEED Statement	967
The STOP OBJECT Statement	968
The SET OBJECT FRAME Statement	968
The SET OBJECT INTERPOLATION Statement	969
The APPEND OBJECT Statement	970
Retrieving Animation Object Information	971
The OBJECT PLAYING Statement	971
The OBJECT LOOPING Statement	971
The OBJECT FRAME Statement	972
The OBJECT SPEED Statement	972
The OBJECT INTERPOLATION Statement	972
The OBJECT SIZE Statement	973
Limbs	974
Summary	975
Solutions	977

Chapter 37 Screen Control

User Control	980
Introduction	980
Selecting an Object	980
The OBJECT SCREEN Statement	982
The PICK OBJECT Statement	983
The GET PICK DISTANCE Statement	984
The PICK VECTOR Statement	985
The PICK SCREEN Statement	986

The OBJECT IN SCREEN Statement	987
Selecting Objects using the Mouse	988
Summary	990
Solutions	991

Chapter 38 Solitaire

Solitaire - The Board Game	994
Introduction	994
The Equipment	994
The Aim	994
The Rules	994
Creating a Computer Version of the Game	994
User Controls	994
Game Responses	995
Screen Layout	995
Media Used	995
Data Structures	996
Adding SetUpScreen()	999
Adding SetUpGame()	1000
Adding CreateBoard()	1001
Adding CreateInternalBoard()	1001
Adding CreateMarbles()	1002
Adding CreateSelector()	1002
Adding SetUpHelp()	1003
Adding GetPlayerMove()	1004
Adding MoveSelector()	1006
Adding SelectMarble()	1007
Adding SelectPit()	1008
Adding IsValidMove()	1008
Adding MoveMarble()	1008
Adding SelectHelpPage()	1009
Using the Mouse	1009
Introduction	1009
Updating the Program	1010
Suggested Enhancements	1013
Solutions	1015

Chapter 39 Advanced Lighting and Texturing

Advanced Lighting and Texturing	1028
Introduction	1028
Surface Reflection	1028
The SET OBJECT AMBIENT Statement	1029
The SET OBJECT DIFFUSE Statement	1030
The SET OBJECT SPECULAR Statement	1030

The SET OBJECT SPECULAR POWER Statement	1031
The SET OBJECT EMISSIVE Statement	1031
The SET OBJECT LIGHT Statement	1034
Mappings	1035
The SET LIGHT MAPPING ON Statement	1035
The SET BUMP MAPPING ON Statement	1038
The SET SPHERE MAPPING ON Statement	1039
The SET BLEND MAPPING ON Statement	1041
The SET CUBE MAPPING ON Statement	1042
The SET ALPHA MAPPING ON Statement	1044
Shadows	1045
The SET SHADOW SHADING ON Statement	1045
The SET SHADOW SHADING OFF Statement	1048
The SET GLOBAL SHADOWS Statement	1048
The SET GLOBAL SHADOW COLOR Statement	1050
The SET GLOBAL SHADOW SHADES Statement	1050
Positioning Shadows	1051
The SET SHADOW POSITION Statement	1051
Shadows and Models	1052
Other Shading Methods	1054
The SET CARTOON SHADING ON Statement	1054
The SET RAINBOW SHADING ON Statement	1056
The SET REFLECTION SHADING ON Statement	1057
The SET SHADING OFF Statement	1058
Summary	1058
Solutions	1061

Chapter 40

Collisions

Object Collisions	1068
Introduction	1068
Object Collision	1068
The OBJECT HIT Statement	1069
The OBJECT COLLISION Statement	1070
The SET OBJECT COLLISION Statement	1070
The SET GLOBAL COLLISION Statement	1071
How Collision Detection Works	1071
The SHOW OBJECT BOUNDS Statement	1072
The HIDE OBJECT BOUNDS statement	1072
Modifying Collision Detection	1074
The SET OBJECT COLLISION TO SPHERES Statement	1074
The SET OBJECT RADIUS Statement	1074
The OBJECT COLLISION RADIUS Statement	1075
The OBJECT COLLISION CENTER Statement	1075
The SET OBJECT COLLISION TO BOXES Statement	1076
The SET OBJECT COLLISION TO POLYGONS Statement	1076

The MAKE OBJECT COLLISION BOX Statement	1077
The GET OBJECT COLLISION Statement	1080
The DELETE OBJECT COLLISION BOX Statement	1082
The AUTOMATIC OBJECT COLLISION Statement	1082
The INTERSECT OBJECT Statement	1083
Summary	1085
Static Collisions	1087
Introduction	1087
Creating and Using Static Collision Boxes	1087
The MAKE STATIC COLLISION BOX Statement	1087
The GET STATIC COLLISION HIT Statement	1087
The GET STATIC COLLISION Statement	1089
The STATIC LINE OF SIGHT Statement	1093
The STATIC LINE OF SIGHT Coordinates Statement	1095
Static Collision Boxes and the Camera	1096
Summary	1096
Solutions	1098

Chapter 41

Particles

Particles	1102
Introduction	1102
Creating Particles	1102
The MAKE PARTICLES Statement	1102
The HIDE PARTICLES Statement	1103
The SHOW PARTICLES Statement	1104
The DELETE PARTICLES Statement	1104
The POSITION PARTICLES Statement	1104
The POSITION PARTICLE EMISSIONS Statement	1105
The ROTATE PARTICLES Statement	1106
The COLOR PARTICLES Statement	1107
The SET PARTICLE EMISSIONS Statement	1108
The SET PARTICLE VELOCITY Statement	1109
The SET PARTICLE GRAVITY Statement	1110
The SET PARTICLE CHAOS Statement	1110
The SET PARTICLE SPEED Statement	1111
The SET PARTICLE FLOOR Statement	1112
The SET PARTICLE LIFE Statement	1113
The GHOST PARTICLES ON Statement	1113
The GHOST PARTICLES OFF Statement	1114
Retrieving Data on a Particles Object	1114
The PARTICLES EXIST Statement	1114
The PARTICLES POSITION Statement	1115
Particles Statements that use Vectors	1116
The SET VECTOR3 TO PARTICLES POSITION Statement	1116
The SET VECTOR3 TO PARTICLES ROTATION Statement	1116

Summary	1116
Other Types of Particles	1118
Introduction	1118
The Statements	1118
The MAKE SNOW PARTICLES Statement	1118
The MAKE FIRE PARTICLES Statement	1119
Summary	1120
Examples of Using Particles	1121
Introduction	1121
A Roman Candle	1121
A Spaceship	1122
A Dungeon Torch	1122
Solutions	1124

Chapter 42

The Elevators Game

Elevators	1128
Introduction	1128
The Equipment	1128
The Aim	1128
The Rules	1128
Creating a Computer version of the Game	1128
User Controls	1128
Game Responses	1128
Screen Layout	1128
The Board Design	1129
The Media Used	1129
Data Structures	1130
Game Logic	1131
Adding SetUpGame()	1132
Adding InitialiseData()	1134
Adding InitialiseLifts()	1134
Adding InitialiseBoard()	1135
Adding InitialiseVisuals()	1136
Loading Models and Texture Files	1136
Adding LoadBoard()	1137
Adding AddElevators()	1137
Adding LoadPlayerCharacter()	1138
Adding LoadDice()	1138
Adding PositionCameras()	1138
Adding RollDice()	1142
Adding MovePlayer()	1143
Adding UseElevator()	1146
Adding MovePlayerToElevator()	1147
Adding TurnPlayer()	1148
Adding MoveOntoPlatform()	1148

Adding MoveElevator()	1148
Adding MoveOffPlatform()	1149
Adding ReturnElevator()	1150
Adding RepositionCamera()	1150
Fixing the Shortcomings	1151
Fixing RepositionCamera()	1151
Fixing MovePlayer()	1152
Fixing UseElevator()	1153
Fixing MovePlayerToElevator()	1153
Fixing MoveElevator()	1153
Adding EndGame()	1154
Game Review	1154
Solutions	1155

Chapter 43 Handling BSP Models

Binary Space Partitioning	1164
Introduction	1164
Creating a BSP File	1165
Using BSP Files	1165
The LOAD BSP Statement	1165
The SET BSP CAMERA COLLISION Statement	1167
The SET BSP OBJECT COLLISION Statement	1167
The SET BSP CAMERA COLLISION RADIUS Statement	1169
The SET BSP OBJECT COLLISION RADIUS Statement	1169
The SET BSP COLLISION HEIGHT ADJUSTMENT Statement	1170
The SET BSP COLLISION THRESHOLD Statement	1171
The PROCESS BSP COLLISION Statement	1171
The SET BSP COLLISION OFF Statement	1171
The BSP COLLISION HIT Statement	1172
The BSP COLLISION Statement	1172
The SET BSP CAMERA Statement	1173
The DELETE BSP Statement	1173
The SET BSP MULTITEXTURING Statement	1173
Summary	1173
Using a BSP Map	1175
Introduction	1175
The Program	1175
Solutions	1178

Chapter 44 Creating Terrain

Creating Terrain	1180
Introduction	1180
Documented Terrain Statements	1180
The MAKE TERRAIN Statement	1180

The DELETE TERRAIN Statement	1181
The POSITION TERRAIN Statement	1182
The TERRAIN POSITION Statement	1183
The TEXTURE TERRAIN Statement	1183
The GET TERRAIN HEIGHT Statement	1184
The GET TOTAL TERRAIN HEIGHT Statement	1186
The Advanced Terrain Statements	1186
The MAKE OBJECT TERRAIN Statement	1186
The SET TERRAIN HEIGHTMAP Statement	1187
The SET TERRAIN SCALE Statement	1187
The SET TERRAIN TEXTURE Statement	1188
The BUILD TERRAIN Statement	1188
The SET TERRAIN TILING Statement	1189
The SET TERRAIN LIGHT Statement	1190
The SET TERRAIN SPLIT Statement	1191
The GET TERRAIN GROUND HEIGHT Statement	1191
The GET TERRAIN SIZE Statement	1193
The SAVE TERRAIN Statement	1193
The LOAD TERRAIN Statement	1194
Terrains as Objects	1195
Summary	1195
Documented Statements	1195
Undocumented (Advanced Terrain) Statements	1196
Terrain Project	1197
Introduction	1197
Creating the Game	1197
Constants and Global Variables	1198
Adding StartUpGame()	1198
Adding PositionCamera()	1199
Adding CreateScene()	1199
Adding LoadTerrain()	1199
Adding CreateSkyBox()	1200
Adding LoadOcean()	1200
Adding PlaceOrb()	1201
Adding StartGame()	1202
Adding ControlPlayer()	1202
Adding EndGame()	1203
Adding Testing Features	1204
Solutions	1206

Chapter 45

Using Matrices

Matrices	1212
Introduction	1212
Creating a Matrix	1213
The MAKE MATRIX Statement	1213

The RANDOMIZE MATRIX Statement	1214
The UPDATE MATRIX Statement	1214
The SET MATRIX HEIGHT Statement	1215
The GET MATRIX HEIGHT Statement	1217
The GET GROUND HEIGHT Statement	1218
The SET MATRIX WIREFRAME Statement	1219
The MATRIX WIREFRAME STATE Statement	1220
Adding Texture to the Matrix	1220
The PREPARE MATRIX TEXTURE Statement	1220
The FILL MATRIX Statement	1222
The SET MATRIX TILE Statement	1223
The SET TEXTURE TRIM Statement	1226
The SHIFT MATRIX Statement	1227
The MATRIX TILE COUNT Statement	1228
The MATRIX TILES EXIST Statement	1228
Positioning the Matrix in 3D Space	1229
The POSITION MATRIX Statement	1229
The MATRIX POSITION Statement	1230
Matrix Transparency	1231
The GHOST MATRIX ON Statement	1231
The GHOST MATRIX OFF Statement	1232
The SET MATRIX PRIORITY Statement	1232
Lighting the Matrix	1234
The SET MATRIX NORMAL Statement	1234
The SET MATRIX Statement	1235
The MATRIX EXIST Statement	1237
Summary	1238
Solutions	1240

Chapter 46

Manipulating Vertices

Manipulating Vertices	1246
Introduction	1246
The Statements	1246
The LOCK VERTEXDATA FOR MESH Statement	1246
The GET VERTEXDATA VERTEX COUNT Statement	1247
The GET VERTEXDATA POSITION Statement	1248
The SET VERTEXDATA POSITION Statement	1250
The UNLOCK VERTEXDATA Statement	1250
The LOCK VERTEXDATA FOR LIMB Statement	1251
The GET VERTEXDATA NORMALS Statement	1253
The SET VERTEXDATA NORMALS Statement	1254
The GET VERTEXDATA Statement	1255
The SET VERTEXDATA UV Statement	1256
The SET VERTEXDATA DIFFUSE Statement	1257
The GET VERTEXDATA DIFFUSE Statement	1258

Handling More Complex Shapes	1258
The ADD MESH TO VERTEXDATA Statement	1265
More About the Vertex Data Buffer's Structure	1266
The GET VERTEXDATA INDEX COUNT Statement	1267
The GET INDEXDATA Statement	1268
The SET INDEXDATA Statement	1270
The DELETE MESH FROM VERTEXDATA Statement	1271
Summary	1272
Solutions	1274

Chapter 47

Accessing Memory

Accessing Memory	1280
Introduction	1280
Pointers	1280
Creating Pointers in DarkBASIC Pro	1281
Assigning a Value to a Pointer	1281
The MAKE MEMBLOCK Statement	1281
The GET MEMBLOCK PTR Statement	1281
Using a Pointer	1282
Using a Pointer to Return Values from a Function	1283
Larger Memory Blocks	1284
The WRITE MEMBLOCK Statement	1284
The MEMBLOCK Statement	1285
The GET MEMBLOCK SIZE Statement	1286
The DELETE MEMBLOCK Statement	1286
The MEMBLOCK EXIST Statement	1286
The COPY MEMBLOCK Statement	1287
Strings and Memory Blocks	1288
The WRITE MEMBLOCK (to file) Statement	1290
The MAKE FILE FROM MEMBLOCK Statement	1292
The READ MEMBLOCK (from file) Statement	1292
The MAKE MEMBLOCK FROM FILE Statement	1294
Adding a New Top Score to our List	1294
Summary	1295
Media Contents and Memory Blocks	1297
Introduction	1297
Bitmaps and Memory Blocks	1297
The MAKE MEMBLOCK FROM BITMAP Statement	1297
The MAKE BITMAP FROM MEMBLOCK Statement	1399
Mapping a Screen Position to a Memory Block Location	1300
Mapping the Mouse Position to a Memory Block Location	1301
Images and Memory Blocks	1302
The MAKE MEMBLOCK FROM IMAGE Statement	1302
The MAKE IMAGE FROM MEMBLOCK Statement	1302
Sounds and Memory Blocks	1303

The MAKE MEMBLOCK FROM SOUND Statement	1303
The MAKE SOUND FROM MEMBLOCK Statement	1305
3D Objects and Memory Blocks	1306
The MAKE MEMBLOCK FROM MESH Statement	1306
The MAKE MESH FROM MEMBLOCK Statement	1309
The CHANGE MESH FROM MEMBLOCK Statement	1310
Summary	1311
Solutions	1312

Chapter 48 Open Dynamics Engine

Using ODE	1318
Introduction	1318
Basic ODE Statements	1318
The ODE CREATE DYNAMIC BOX Statement	1318
The ODE START Statement	1319
The ODE END Statement	1319
The ODE UPDATE Statement	1319
The ODE SET WORLD GRAVITY Statement	1320
The ODE CREATE STATIC BOX Statement	1321
The ODE CREATE DYNAMIC SPHERE Statement	1322
The ODE CREATE DYNAMIC CYLINDER Statement	1322
The ODE CREATE DYNAMIC TRIANGLE MESH Statement	1324
The ODE SET WORLD STEP	1325
The ODE CREATE STATIC TRIANGLE MESH Statement	1325
The ODE SET WORLD ERP Statement	1326
The ODE SET WORLD CFM Statement	1327
The ODE SET CONTACT FDIR1 Statement	1328
The ODE SET LINEAR VELOCITY Statement	1328
The ODE SET ANGULAR VELOCITY Statement	1331
The ODE SET BODY ROTATION Statement	1332
The ODE SET BODY MASS Statement	1332
The ODE DESTROY OBJECT Statement	1334
The ODE GET BODY LINEAR VELOCITY Statement	1335
The ODE GET BODY HEIGHT Statement	1335
The ODE COLLISION MESSAGE EXISTS Statement	1336
The ODE COLLISION GET MESSAGE Statement	1336
The ODE GET OBJECT Statement	1336
The ODE GET OBJECT VELOCITY Statement	1337
The ODE GET OBJECT ANGULAR VELOCITY Statement	1338
The ODE ADD FORCE Statement	1338
Surface Contact Statements	1340
Summary	1343
Solutions	1345

3D Vectors	1350
Introduction	1350
A Mathematical Description of 3D Vectors	1350
What is a 3D Vector in DarkBASIC Pro?	1351
Why do we need 3D Vectors?	1351
3D Vector Statements	1351
The MAKE VECTOR3 Statement	1351
The SET VECTOR3 Statement	1352
Retrieving Data from a 3D Vector	1352
The DELETE VECTOR3 Statement	1353
The COPY VECTOR3 Statement	1353
The MULTIPLY VECTOR3 Statement	1354
The SCALE VECTOR3 Statement	1354
The DIVIDE VECTOR3 Statement	1355
The LENGTH VECTOR3 Statement	1355
The SQUARED LENGTH VECTOR3 Statement	1356
The ADD VECTOR3 Statement	1356
The SUBTRACT VECTOR3 Statement	1357
The DOT PRODUCT VECTOR3 Statement	1357
The NORMALIZE VECTOR3 Statement	1358
The IS EQUAL VECTOR3 Statement	1359
The MAXIMIZE VECTOR3 Statement	1359
The MINIMIZE VECTOR3 Statement	1360
The CROSS PRODUCT VECTOR3 Statement	1360
Summary	1361
4D Vectors	1363
Introduction	1363
Matrices	1365
Introduction	1365
Matrix Statements	1365
The MAKE MATRIX4 Statement	1365
The SET IDENTITY MATRIX4 Statement	1366
The IS IDENTITY MATRIX4 Statement	1366
Other Matrix Assignment Statements	1367
The COPY MATRIX4 Statement	1367
The IS EQUAL MATRIX4 Statement	1367
The ADD MATRIX4 Statement	1368
The SUBTRACT MATRIX4 Statement	1368
The DIVIDE MATRIX4 Statement	1368
The MULTIPLY MATRIX4 Statement	1369
The INVERSE MATRIX4 Statement	1370
The SCALE MATRIX4 Statement	1370
The TRANSLATE MATRIX4 Statement	1371
The ROTATE MATRIX4 Statement	1371

The TRANSPOSE MATRIX4 Statement	1372
The DELETE MATRIX4 Statement	1372
Summary	1372
Solutions	1374

Chapter 50 Shaders

Shaders and FX Files	1376
Introduction	1376
Vertex Shader	1376
Pixel Shader	1376
FX Files	1377
Graphics Card Check Statements	1377
The GET MAXIMUM VERTEX SHADER VERSION Statement	1377
The GET MAXIMUM PIXEL SHADER VERSION Statement	1377
FX Statements	1378
The LOAD EFFECT Statement	1378
The EFFECT EXIST Statement	1378
The PERFORM CHECKLIST FOR EFFECT ERRORS Statement	1379
The SET OBJECT EFFECT Statement	1379
The SET EFFECT ON Statement	1380
The DELETE EFFECT Statement	1381
The SET LIMB EFFECT Statement	1381
The PERFORM CHECKLIST FOR EFFECT VALUES Statement	1382
The SET EFFECT CONSTANT Statement	1383
The SET EFFECT TECHNIQUE Statement	1383
The SET EFFECT TRANSPOSE Statement	1384
Vertex Shader Statements	1383
The CREATE VERTEX SHADER FROM FILE Statement	1383
The SET VERTEX SHADER ON Statement	1385
The SET VERTEX SHADER OFF Statement	1385
The DELETE VERTEX SHADER Statement	1385
Other Vertex Shader Statements	1386
Pixel Shader Statements	1386
Summary	1386
FX Files	1386
Shader Files	1387
Solutions	1388

Chapter 51 Network Programming

Networked Games	1390
Introduction	1390
Hardware Requirements	1390
Getting Started	1390
The PERFORM CHECKLIST FOR NET CONNECTIONS Statement	1391

TCP/IP	1392
The SET NET CONNECTION Statement	1392
The CREATE NET GAME Statement	1394
Writing Code for the Client Machine	1395
The PERFORM CHECKLIST FOR NET SESSIONS Statement	1395
The JOIN NET GAME Statement	1396
The PERFORM CHECKLIST FOR NET PLAYERS Statement	1397
Using a Single Machine as Both Host and Client	1498
Combining the Host/Client Requirements	1499
Communicating	1401
The SEND NET MESSAGE Statement (Version 1)	1401
The GET NET MESSAGE Statement	1401
The NET MESSAGE EXISTS Statement	1402
The NET MESSAGE Statement (Version 1)	1402
The NET MESSAGE PLAYER FROM Statement	1403
The NET MESSAGE PLAYER TO Statement	1403
The SEND NET MESSAGE Statement (Version 2)	1404
The NET MESSAGE Statement (Version 2)	1405
The NET MESSAGE TYPE Statement	1406
The NET BUFFER SIZE Statement	1408
Session Dynamics	1409
The NET PLAYER CREATED Statement	1409
The NET PLAYER DESTROYED Statement	1409
The NET GAME NOW HOSTING Statement	1411
The FREE NET GAME Statement	1411
The CREATE NET PLAYER Statement	1412
The FREE NET PLAYER Statement	1412
The NET GAME EXISTS Statement	1413
The NET GAME LOST Statement	1413
Summary	1413
A Networked Game	1415
Introduction	1415
A Non-Networked Version	1415
Program Data	1415
Game Logic	1416
Adding SetUpPlayerDetails()	1417
Adding SetUpScreen()	1417
Adding SetUpBoard()	1417
Adding GetMove()	1417
Adding GetMyMove()	1418
Adding GetSquare()	1418
Adding InRange()	1418
Adding GetOpponentsMove()	1419
Adding CheckForWin()	1419
Adding the Other Search Routines	1420
Adding EndGame()	1422

Networking the Game	1423
Updating the main section	1423
Adding WaitForSecondPlayer()	1423
Adding NumberOfPlayers()	1423
Modifying the Call to SetUpPlayerDetails()	1424
Modifying GetMyMove()	1424
Modifying GetOpponentsMove()	1424
Modifying EndGame()	1425
A Complete Listing	1425
Solutions	1431

Chapter 52 Using File Transfer Protocol

Internet File Transfers	1436
Introduction	1436
The Instructions	1436
The FTP CONNECT Statement	1436
The GET FTP FAILURE Statement	1436
The GET FTP ERROR\$ Statement	1437
The GET FTP STATUS Statement	1437
The FTP SET DIR Statement	1438
The GET FTP DIR\$ Statement	1438
The FTP FIND FIRST Statement	1438
The FTP FIND NEXT Statement	1439
The GET FTP FILE TYPE Statement	1439
The GET FTP FILE NAME\$ Statement	1439
The GET FTP FILE SIZE Statement	1439
The FTP DISCONNECT Statement	1440
The FTP GET FILE Statement	1440
The FTP PROCEED Statement	1441
The GET FTP PROGRESS Statement	1442
The FTP TERMINATE Statement	1442
The FTP DELETE FILE statement	1442
The FTP PUT FILE Statement	1443
Summary	1443

Chapter 53 Dynamic Link Libraries

Creating New DBPro Statements	1446
Introduction	1446
A Dynamic Link Library (DLL)	1446
Creating a DLL	1446
Starting Up Visual Studio	1446
Adding the Code for New Statements	1448
Adding a String Table	1449
Constructing the Caption	1450

Adding the New Statements to DarkBASIC Pro	1451
Adding Help	1452
Adding More New Commands	1456
Functions that Return Real Values	1456
Functions that Return Strings	1456
More String Handling Functions	1459
Summary	1460
Using Standard DLLs	1462
Introduction	1462
The LOAD DLL Statement	1463
The DLL EXIST Statement	1463
The CALL DLL Statement	1463
The DLL CALL EXIST Statement	1464
The DELETE DLL Statement	1464
Summary	1465
Solutions	1467

Acknowledgements

I would like to thank all those who helped me prepare the final draft of this book.

In particular, Virginia Marshall who proof-read the original script and Michael Kerr who did an excellent job of checking the technical contents. Mark Armstrong researched all the difficult bits for me and produced almost as much in the way of notes as is in this final text.

Any errors that remain are probably due to the usual extra paragraphs I added after all the proof-reading was complete!

Thanks also to The Game Creators Ltd for producing an excellent piece of software - DarkBASIC Professional.

Many of the 3D models and textures are from The Game Creators Dark Matter 1 package and used with their kind permission.

Finally, thank you to every one of you who has bought this book. Any constructive comments would be most welcome.

Email me at *alistair@digital-skills.co.uk*.

Introduction

Welcome to the second volume of a book that I hope is a little different from any other you've come across before. Instead of just telling you about software design and programming, it makes you get involved. There's plenty of work for you to do since the book is full of exercises - most of them programming exercises - but you also get a full set of solutions, just in case you get stuck!

If you've worked your way through Volume 1, then you should have gained a good grounding in, not only DarkBASIC Pro, but also professional programming skills.

Most of Volume 2 is dedicated to 3D graphics but there are a few other interesting topics such as network programming and how to create your own DarkBASIC commands.

Learn by Doing

The only way to become a programming expert is to practice. No one ever learned any skill by just reading about it! Hence, this is not a text book where you can just sit back in a passive way and read from cover to cover whilst sitting in your favourite chair. Rather it is designed as a teaching package in which you will do most of the work.

The tasks embedded in the text are included to test your understanding of what has gone before and as a method of helping you retain the knowledge you have gained. It is therefore important that you tackle each task as you come to it. Also, many of the programming exercises are referred to, or expanded, in later pages so it is important that you are familiar with the code concerned.

What You Need

You'll obviously need a PC and a copy of DarkBASIC Pro.

At this stage you'll also need some programming skills and a basic knowledge of DarkBASIC Pro.

How to Get the Most out of this Text

Experience has shown that readers derive most benefit from this material by approaching its study in an organised way. The following strategy for study is highly recommended:

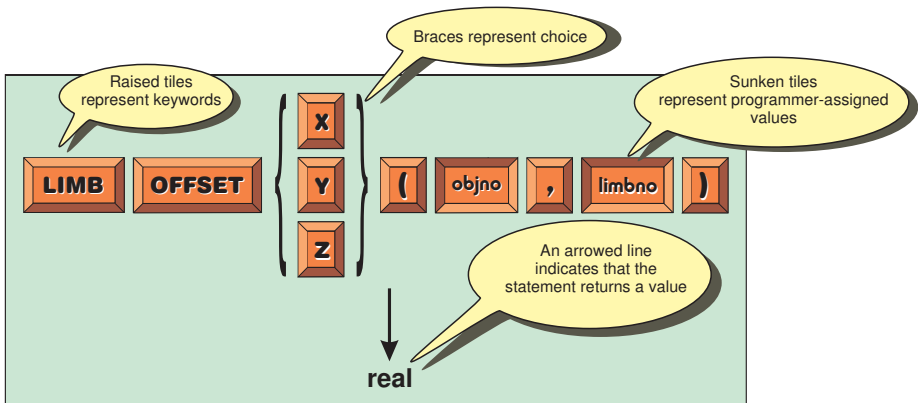
1. Read a chapter or section through without taking notes or worrying too much about topics that are not immediately clear to you. This will give you an overview of the contents of that chapter/section.
2. Re-read the chapter. This time take things slowly; make notes and summaries of the material you are reading (even if you understand the material, making notes helps to retain the facts in your long-term memory); re-read any parts you are unclear about.
3. Embedded in the material are a series of activities. Do each task as you reach it (on the second reading). These activities are designed to test your knowledge and understanding of what has gone before. Do not be tempted to skip over them, promise to come back to them later, or to

make only a half-hearted attempt at tackling them before looking up the answer (there are solutions at the end of each chapter). Once you have attempted a task, look at the solution given. Often there will be important points emphasised in the solution which will aid higher understanding.

4. As you progress through the book, go back and re-read earlier chapters, since you will often get something new from them as your knowledge increases.

Syntax Diagrams

The format of each statement is explained using a syntax diagram. Raised tiles represent keywords of the language while sunken tiles are parts of the statement for which you are free to create your own values. Parts within square brackets are optional while braces represent a choice of options. Statements that return a value show this using an arrowed line and the type of value returned.



Line Continuation Symbol

Occasionally, a single programming instruction has to be split over two or more lines because of limited page width. In such cases the second line (and subsequent lines) begins with the \hookrightarrow symbol. For example, the instruction

```
POSITION OBJECT 2,OBJECT POSITION X(2),OBJECT POSITION Y(2)-0.1,OBJECT POSITION Z(2)+0.1
```

might appear as

```
POSITION OBJECT 2,OBJECT POSITION X(2),  
 $\hookrightarrow$ OBJECT POSITION Y(2)-0.1,OBJECT POSITION Z(2)+0.1
```

In such cases you should enter the code as a single line when creating a DarkBASIC Pro program.



3D Concepts and Terminology

3D Coordinate System

3D Primitives

3D Vectors

Cameras

Lights

Vertex and Surface Normals

Rotation

Textures

The Major Planes in 3D

Vertices, Edges and Polygons

Wireframe Models

World Units

The 3D World

Introduction

Welcome to the world of 3D. Of course, we can create great games in 2D - many people still consider 2D games like Space Invaders and Pac-Man to be some of the best games ever invented - but for sheer eye candy you really can't beat 3D.

In this chapter we'll get a broad view of the 3D world created by computers. We'll cover the basic concepts and define some of the terms. Many of these concepts will be explained in greater detail in later chapters as we discover how DarkBASIC Pro implements many of these ideas.

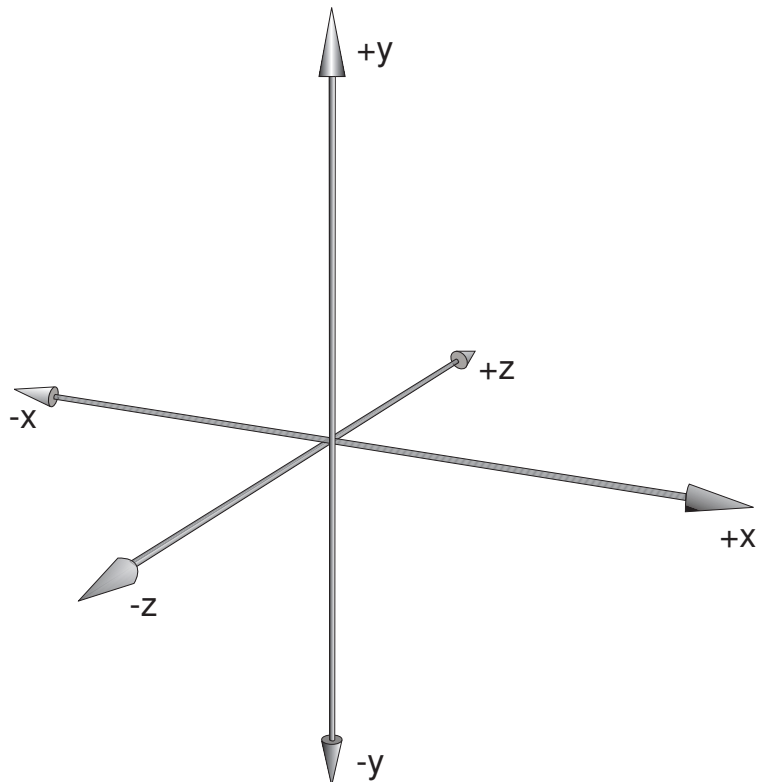
The Coordinate System

Axes

In a 3D world, just as in a 2D one, we need to identify the position of any point within that world. This we do using three axes (known as **world axes**) for reference. As before, we need x and y axes for width and height, but this time we also need a z axis to measure depth (see FIG-30.1).

FIG-30.1

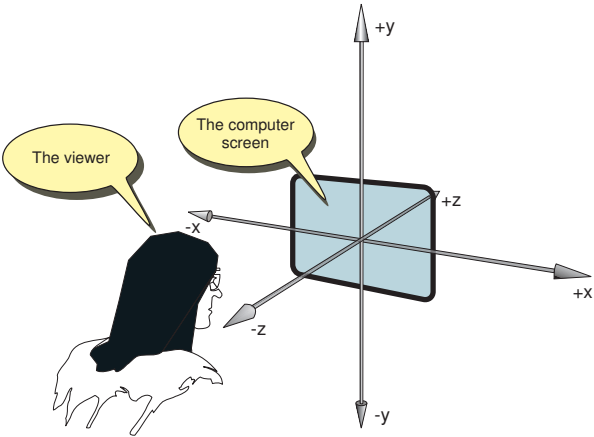
The Axes used in 3D



In the figure above, the axes have been skewed slightly to give a better perspective. In reality the x -axis runs across the screen, the y -axis runs up and down, and the z -axis points directly out of the screen ($-z$) and into the screen ($+z$) (see FIG-30.2).

FIG-30.2

3D Axes and the Viewer

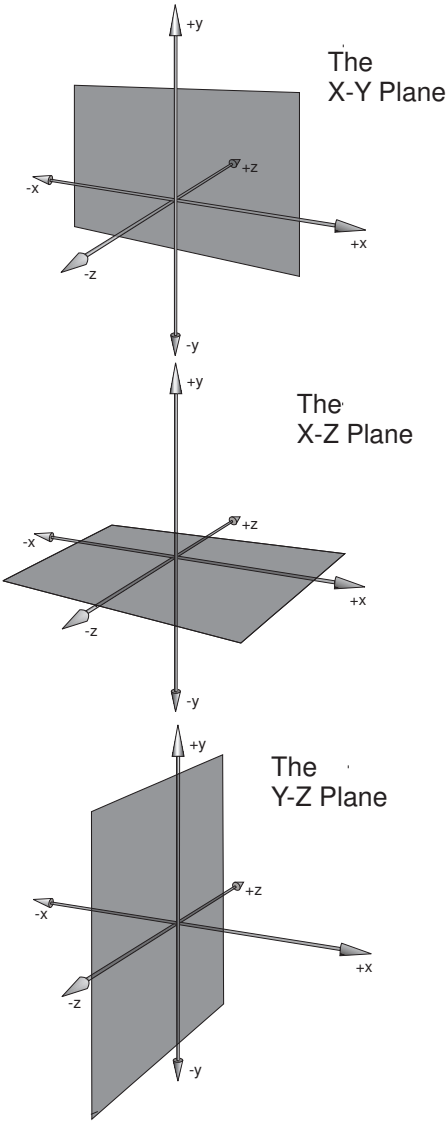


Planes

In mathematics, a **plane** is a flat surface with only two dimensions. 3D space has three main planes: the X-Y plane, the X-Z plane and the Y-Z plane (see FIG-30.3).

FIG-30.3

The Main 3D Planes



The X-Y plane has the x and y axes passing through its centre and, like every plane, expands to infinity in all directions. The X-Z plane has the x and z axes at its centre, and the Y-Z plane has the y and z axes at its centre.

These three planes are important since each divides space into two equally sized areas. The X-Y plane splits space with one half to the front, the other half to the back. The X-Z planes splits space into above and below sections, and the Y-Z plane splits space into left and right sections.

With all three planes in place, space is split into eight equally-sized sections. Each of these sections is known as an **octant**.

Of course, not all planes lie on axes; there are an infinite number of planes, some parallel to the main planes, others at angles to those planes, but it is the main planes that will be useful in many of the calculations required when determining the position of an object in 3D space.

Points

To specify the position of a point in 3D space we state its distance from the origin along all three axes in the order, x, y, z (see FIG-30.4).

FIG-30.4

Determining the Position of a Point in 3D Space

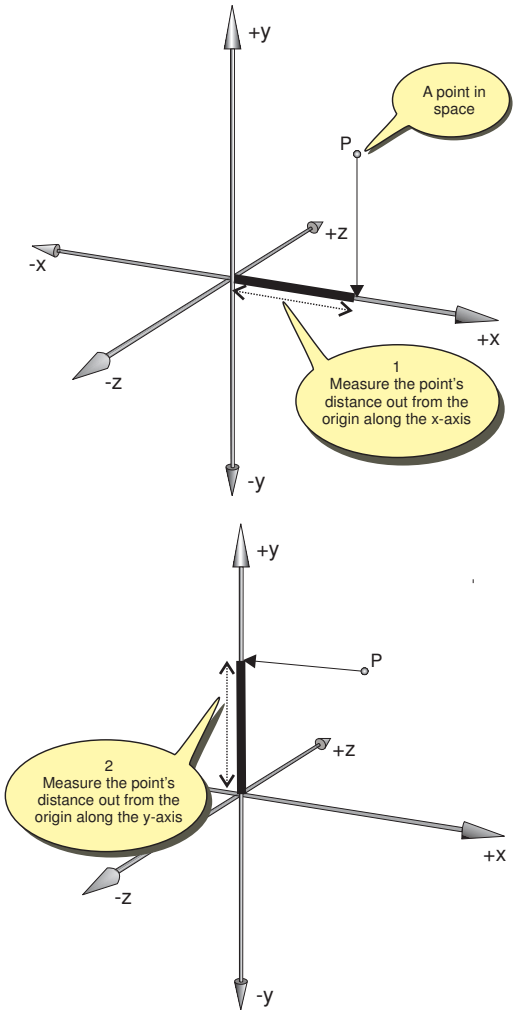
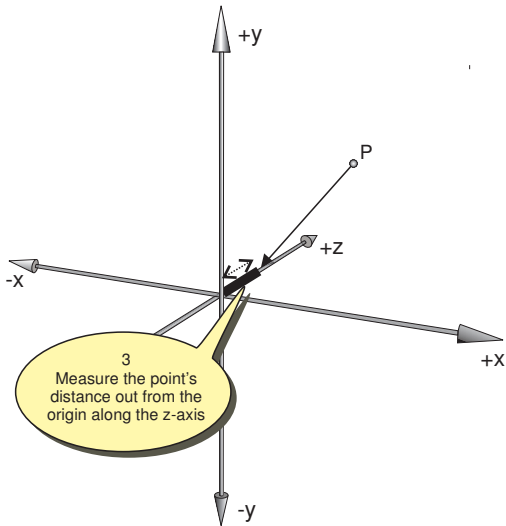


FIG-30.4
(continued)

Determining the Position of a
Point in 3D Space



We might, for example, state that point *p* is at the position (8,12,5) meaning that point *p* is 8 units along the x-axis, 12 units along the y-axis and 5 units along the z-axis.

World Units

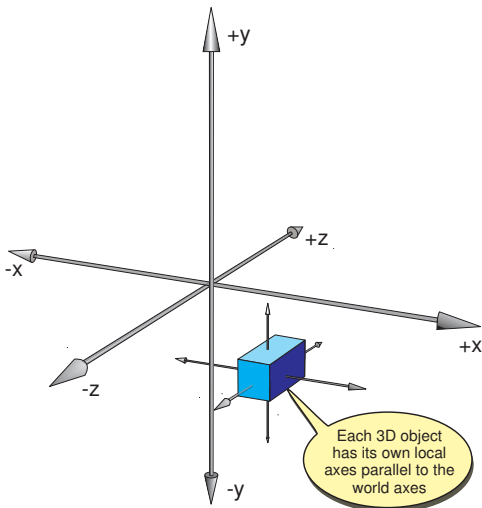
Distances are measured in units. These units have no relationship to real-life measurements such as centimetres or inches. Instead, objects are constructed in such a way as to be the correct size relative to other objects. For example, if we make a human character 6 units high, then a simple house might be 18 to 25 units high. Of course, if you wish, you can think of 1 unit being the equivalent of a real distance. The scale you choose will depend on the context; when creating a world with an ant as the main character, 1 unit might be equivalent to a millimetre, while a truly interstellar game might make 1 unit equivalent to 1 light year.

Local Axes

Every 3D object we create has its own **local axes**. These axes are (initially, at least) aligned to the world axes. FIG-30.5 shows a cuboid and its local axes.

FIG-30.5

Each Object has its
Own Local Axes

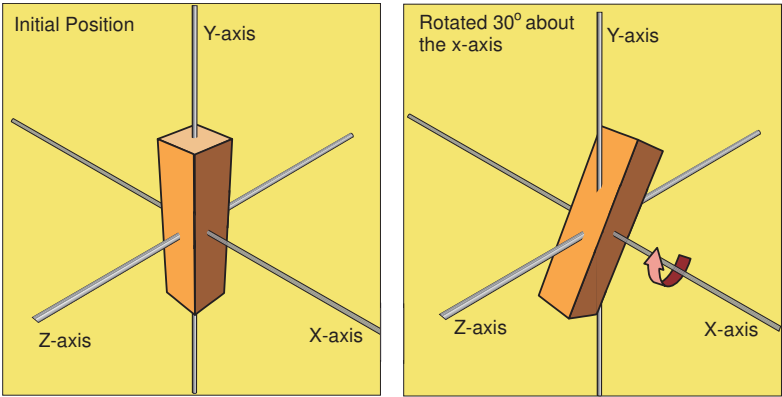


Rotation

An object can be made to rotate about its own, **local**, axes. In DarkBASIC Pro rotation is measured in degrees. For example, we might rotate an object 30 degrees about its x-axis as shown in FIG-30.6.

FIG-30.6

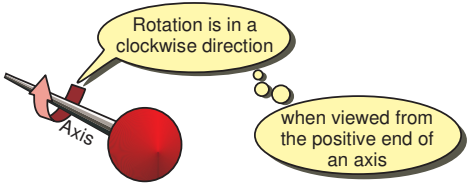
A Cuboid is Rotated 30° about the x-axis



Rotation is performed in a clockwise direction when viewed down the positive end of an axis (see FIG-30.7).

FIG-30.7

Clockwise Rotation



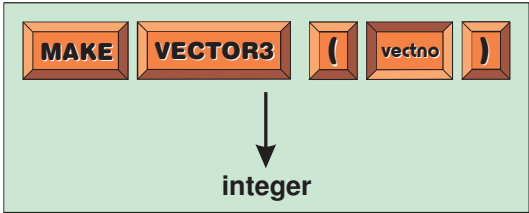
By specifying a negative angle of rotation, an object will rotate anti-clockwise.

3D Vectors

Although the purpose of this chapter is to describe basic 3D concepts, it's worth mentioning that DarkBASIC Pro allows the creation of a 3-element vector specifically for storing the coordinates of a point in 3D space. The vector is created using the MAKE VECTOR3 statement which has the format shown in FIG-30.8.

FIG-30.8

The MAKE VECTOR3 Statement



In the diagram:

vectno

is an integer value giving the ID to be assigned to the 3D vector being created.

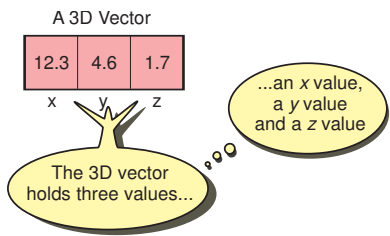
The statement returns 1 if the vector is created successfully; otherwise zero is returned. Usually we won't worry about the value returned and can create a 3D vector with a statement such as:

```
result = MAKE VECTOR3(1)
```

We can visualise a 3D vector object as shown in FIG-30.9.

FIG-30.9

A 3D Vector



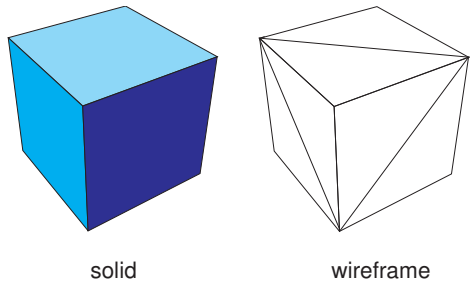
Many of the DarkBASIC Pro statements we'll encounter later make use of 3D vectors for storing results, so it's useful to give you this quick grounding in them at this early stage. We'll learn more on this subject in a later chapter.

Object Terminology

Just as 2D has a few basic shapes such as a line, a circle, a triangle and a rectangle, so we have a set of basic shapes (known as **primitives**) in 3D. These include the sphere, cylinder, cone, and cube. In FIG-30.10 we see an example of a cube.

FIG-30.10

A Cube - An Example of a Primitive

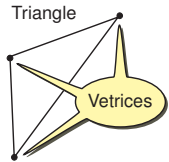


The cube is shown in two ways: **solid**, with shading caused by the light falling on its surface, and **wireframe** showing how the cube is constructed.

Polygon is the term used for a many-sided enclosed area. The simplest polygon (that is, the one with the least sides) is the triangle. The point where two lines of a polygon meet is known as a **vertex**. A triangle has three vertices (see FIG-20.11).

FIG-30.11

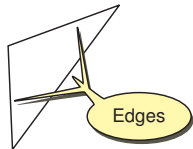
The Vertices of a Triangle



The line between two vertices is known as an **edge** (see FIG-30.12).

FIG-30.12

Edges



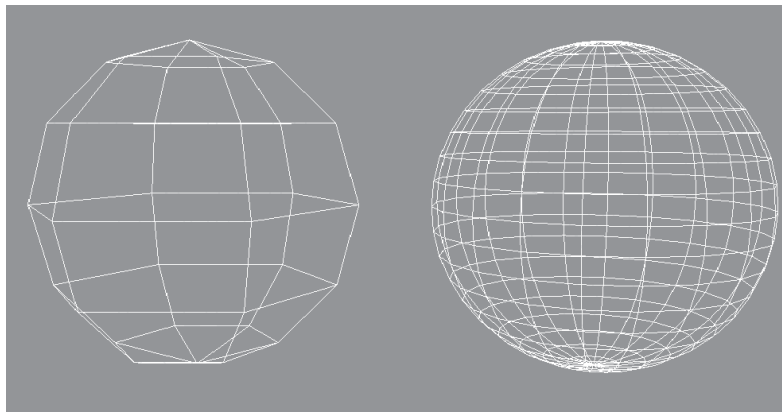
Every 3D shape in a game is constructed from polygons (normally triangles), as you can see from the wireframe version of the cube shown in FIG-30.10.

The greater the number of polygons used to create an object, the more detailed and realistic it will appear (see FIG-30.13). But there is a price to pay for greater detail

- higher processing requirements. As you increase the number of polygons that go to make up the objects in your scene, the harder your processor and video card need to work. Ask too much of your hardware, and screen updating will slow down. The number of times the screen is redrawn in one second is known as the **frame rate** and is quoted in **frames per second** (fps). If the frame rate falls much below about 20 fps, then your eyes will become aware of the screen refreshing and the picture will become jerky.

FIG-30.13

Varying the Polygons in a Sphere



A Sphere with Few Polygons

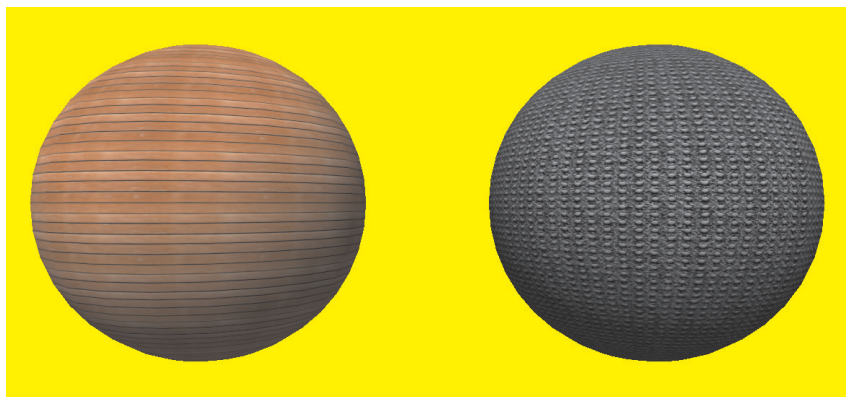
A Sphere with Many Polygons

Textures

In solid mode (as opposed to wireframe), a 3D object has a bland grey surface, but we can use an image wrapped around that object to give it a greater reality. By wrapping the image of riveted steel plate round a sphere, we can create the illusion of a metal ball. Wrap an image of wooden planks round the same sphere and we create a wooden ball (see FIG-30.14).

FIG-30.14

Adding Texture to a 3D Object



Images with an Alpha Channel

The image used to texture an object can be one of many different formats. For example, JPG and BMP files are often used, but sometimes we will see images stored in the PNG or TGA format.

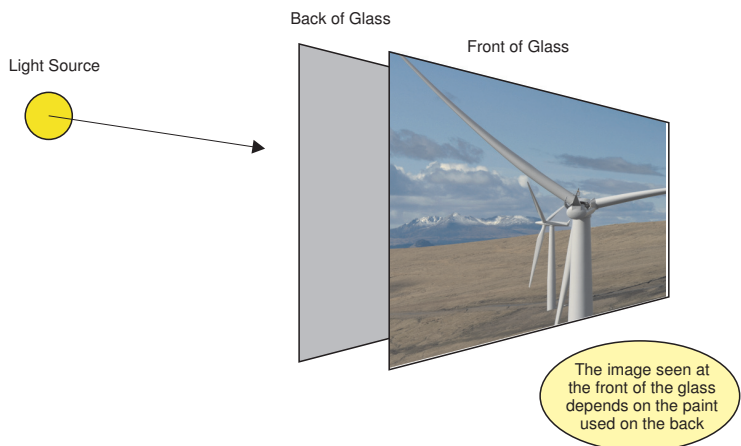
PNG and TGA files are amongst those formats capable of embedding an **alpha channel** within the image. An alpha channel affects how visible an image is and is probably best explained with an analogy.

Imagine you've just painted an image on a piece of glass and that the light illuminating the picture comes from behind the glass (see FIG-30.15) - like looking

out through a church's stained-glass window.

FIG-30.15

Perceived Image
Depends on the Backing

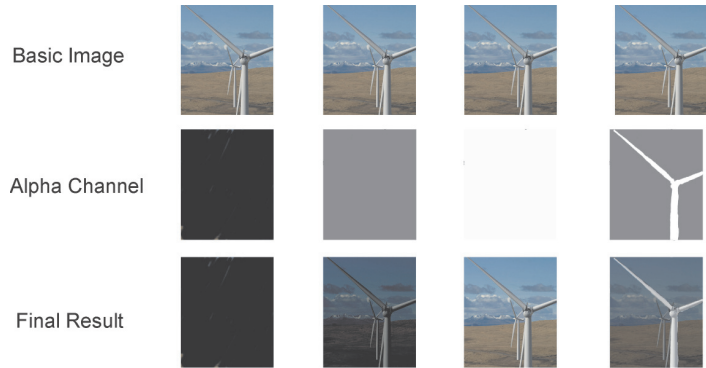


If we were to paint the back of the glass black, no light would get through and we wouldn't see the picture. If we used grey paint rather than black, then some light would get through. If we painted a pattern on the back of the glass using a mixture of black, dark grey, and light grey paint, the image would appear to have bright, dull and black areas depending on the paint on the back of the image.

This is how the alpha channel of an image works. As well as the basic red, green and blue elements (or **channels**) that go to make up the image, a fourth, alpha, channel is added. This is just another layer to the image which can only be shaded using greyscale colours (white through to black). Where black is used, the original image is unseen; where white is used the image appears at normal brightness (this is where the glass analogy falls down since it would be at its brightest with no paint on the back of the glass). Shades of grey give varying degrees of image brightness. FIG-30.16 shows original images, alpha channels, and the overall effects created.

FIG-30.16

Using an Alpha Channel



Cameras

The real world is a vast place, but with the help of television we can view any part of it - all we need is a camera. What the TV camera broadcasts we see on our screens. Move the camera and we see a different part of the world.

This is exactly how the 3D world we create within the computer works; what we see on the computer screen is the output from a virtual camera. The camera can be moved, just like a real camera, revealing different parts of our new 3D world. We can zoom the camera in or out allowing us to enlarge a distant object or show everything within a small space.

We can even use several cameras, switching between each to change what the user is seeing on the screen. Unlike real life, there's never any chance of seeing a camera in the view produced by a second camera - all virtual cameras are invisible!

DarkBASIC Pro creates and positions a single camera automatically at the start of every program that uses 3D objects. The exact position of the camera depends on the positioning of the 3D objects, since the camera normally places itself in order to see the objects that have been created. However, as the programmer, you can take complete control of the camera and thereby determine just exactly what appears on the screen.

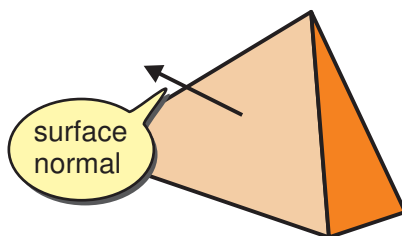
Lights

We can even set up the lights we want to use to illuminate our new world - just like placing lights on a movie set. By positioning various types of lights in just the correct positions, we can create any type of atmosphere we want - from dark and mysterious to bright and sunny. Like cameras in the 3D world, the lights are invisible, but the effects they create are not!

To help calculate the effect of lights on the individual polygons of a 3D object, a set of **normals** are maintained by the object. A **surface normal** is a vector coming from the centre of a polygon, perpendicular to the surface of the polygon. Every polygon within an object has an associated surface normal (see FIG-30.17).

FIG-30.17

Surface Normals



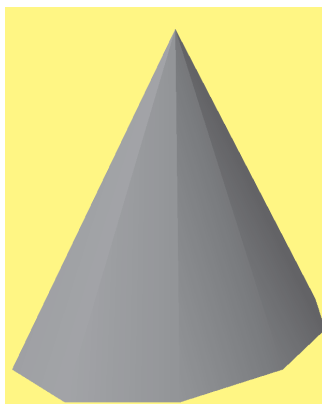
Normals are stored as mathematical expressions and are not part of the visible structure of the model.

In principal every polygon can have two surface normals: one on the top side and one on the bottom. However, often models only use a single normal - on the side facing outwards.

When using surface normals to calculate how an object should be lit, we sometimes get a rather faceted appearance, with an obvious jump in shading from one polygon to the next (see FIG-30.18).

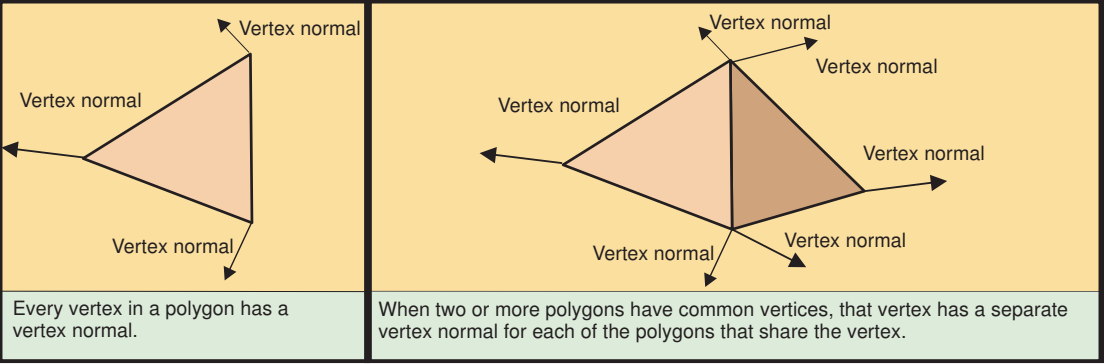
FIG-30.18

Visible Polygons



To solve this, **vertex normals** may be used. A vertex normal is created at every vertex of a polygon (see FIG-30.19).

FIG-30.19 Vertex Normals

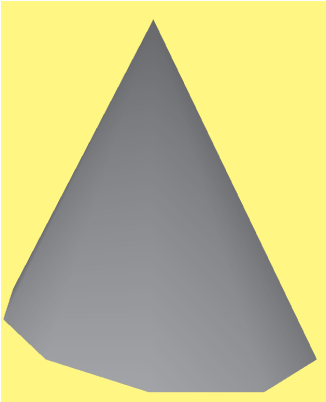


These vertex normals are calculated from the values of the two edges which meet at that vertex.

Using vertex normals creates a smoother lighting effect, but requires more calculations. You can see the effect produced in FIG-30.20.

FIG-30.20

Polygon Smoothing



Activity 30.1

Load and run the program *basic3D.exe*. This will demonstrate some of the basic concepts covered in this chapter.

(You can download this program, and all other files used in this text from www.digital-skills.co.uk)

As we'll see in the chapters that follow, DarkBASIC Pro has literally hundreds of commands designed to help us create a 3D world and manipulate the objects in that world.

Summary

- The 3D world uses three axes: x, y and z.
- 3D space is split into eight octants by the X-Y, X-Z and Y-Z planes.

- Space within the 3D world is measured in world units. These do not relate to real world units.
- A point in 3D space is defined by its distance along each of the axes.
- 3D objects have their own local axes.
- 3D objects can be rotated about their own local axes.
- Rotations are measured in degrees.
- Rotation is in a clockwise direction (as viewed from the positive end of the axis of rotation).
- DarkBASIC Pro provides 3D vector objects in which the coordinates of a point in 3D space can be stored.
- 3D objects are constructed from polygons.
- The simplest polygon is the triangle.
- The end of a line within a polygon is known as a vertex.
- The line between two vertices is known as an edge.
- More detailed objects require more polygons.
- Increasing the number of polygons used in a scene increases the load on the computer.
- When faced by a heavy load, the computer will output at a reduced frame rate.
- Images can be used to texture a 3D shape to increase realism.
- Some images can contain alpha channels which effect lightness when the image is used to texture a surface.
- Virtual cameras determine which parts of the 3D world are shown on the screen.
- Lights can be added to a scene to help create the desired atmosphere.
- The effects of lights on a surface are calculated using surface normals or vertex normals.
- Every polygon has an associated surface normal.
- A surface normal is a vector at right angles to its polygon.
- Using surface normals to calculate shading can result in a patchy effect.
- Every vertex of a polygon has an associated vertex normal.
- Vertex normals may be used to create smoother shading effects, but at the cost of more complex calculations.

Absolute and Relative Object Movement

Global and Local Axes

Creating 3D Primitives

Culling

Deleting 3D Primitives

Duplicating 3D Objects

Merging Objects

Pointing an Object in a Specific Direction

Positioning 3D Objects

Retrieving 3D Object Data

Rotating 3D Objects

Resetting Local Axes

Resizing 3D Objects

Showing and Hiding 3D Objects

Wireframe Mode

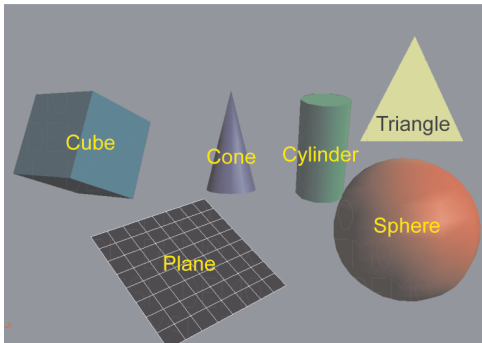
3D Primitives

Introduction

DarkBASIC Pro contains several statements for creating and manipulating 3D primitives such as spheres, cones and cubes. These statements are explained in detail below. A sample of the possible shapes is shown in FIG-31.1.

FIG-31.1

The 3D Shapes that can
be Created in
DarkBASIC Pro



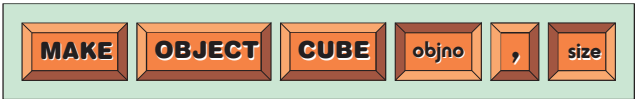
Creating a Cube

The MAKE OBJECT CUBE Statement

To create a cube on the screen, we use the MAKE OBJECT CUBE statement. Like sprites, every 3D object created must be given an identifying integer value (its ID). No two 3D objects within a program can be assigned the same ID. The size of the cube is also defined in this statement, which has the format shown in FIG-31.2.

FIG-31.2

The MAKE OBJECT
CUBE Statement



In the diagram:

- objno* is an integer value giving the ID to be assigned to the cube.
- size* is a real value specifying the width, height and depth of the cube. This value is given in world units.

A typical usage of this statement might be:

```
MAKE OBJECT CUBE 1, 10
```

This would create a cube (with ID 1) which is 10 units wide, by 10 units high, by 10 units deep. FIG-31.3 shows a screen shot of the resulting cube.

FIG-31.3

A Cube in DarkBASIC Pro



This may not look too impressive as a 3D object, but that's because we're looking at the cube straight on and therefore can only see the front face of the object.

The cube shown above was created using the program given in LISTING-31.1.

LISTING-31.1

Creating a Cube

Statements such as COLOR BACKDROP and BACKDROP ON were covered in Volume 1.

```
REM *** Set display resolution and backdrop ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP 0
BACKDROP ON

REM *** Make the cube ***
MAKE OBJECT CUBE 1,10

REM *** End program ***
WAIT KEY
END
```

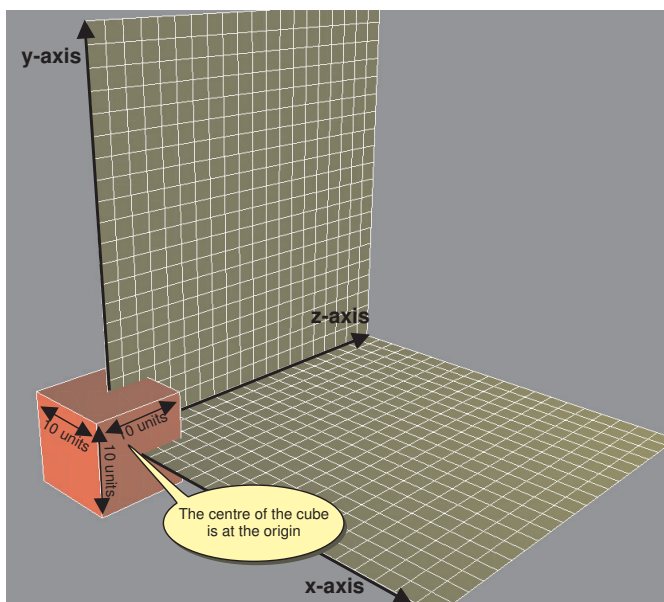
Activity 31.1

Type in the program in LISTING-31.1 (*object3D01.dbpro*) and check that you get the same display as shown above.

When any of the 3D primitives is first created, its centre is positioned at the origin. FIG-31.4 shows a model of what has been created by the program in LISTING-31.1. The 3 axes and parts of the XZ and YZ planes have been included to give a clearer picture of how the cube is positioned.

FIG-31.4

How the Cube is
Positioned by the
Program



Creating Other Primitives

DarkBASIC Pro has a set of similar MAKE statements to create other basic 3D shapes. Like the cube, all of these objects are initially positioned with their centres at the origin. These statements are described below.

The MAKE OBJECT BOX Statement

The MAKE OBJECT BOX statement is similar to the MAKE OBJECT CUBE statement, but allows the three dimensions of the object to be set separately. The statement has the format shown in FIG-31.5.

FIG-31.5

The MAKE OBJECT
BOX Statement



In the diagram:

- objno* is an integer value giving the ID to be assigned to the box being created. No other 3D object in the program can be assigned the same value.
- w* is a real value giving the width (x-dimension) of the box.
- h* is a real value giving the height (y-dimension) of the box.
- d* is a real value giving the depth (z-dimension) of the box.

For example, the line

```
MAKE OBJECT BOX 2, 10, 3.7, 12
```

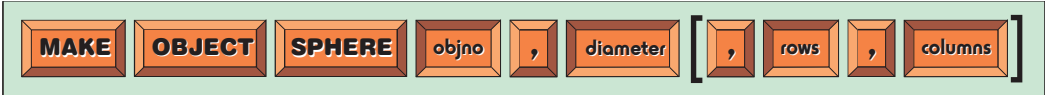
would create a box with ID 2 which is 10 units wide, by 3.7 units high, by 12 units deep.

The MAKE OBJECT SPHERE Statement

The MAKE OBJECT SPHERE statement creates a sphere of a specified diameter but offers extra options. The statement has the format shown in FIG-31.6.

FIG-31.6

The MAKE OBJECT SPHERE Statement



In the diagram:

- objno* is an integer value giving the ID assigned to the sphere being created.
- diameter* is a real number representing the diameter of the sphere.
- rows* is an integer value specifying the number of *lines of latitude* making up the sphere.
- columns* is an integer value specifying the number of *lines of longitude* making up the sphere.

The statement

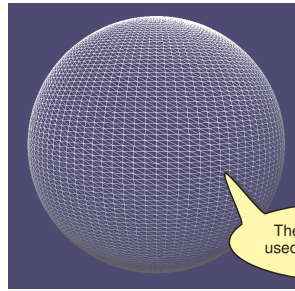
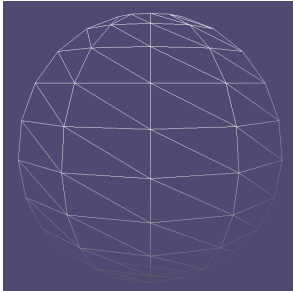
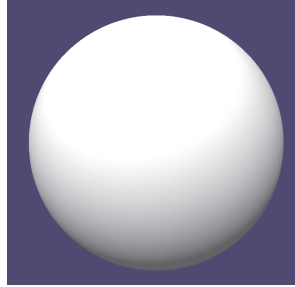
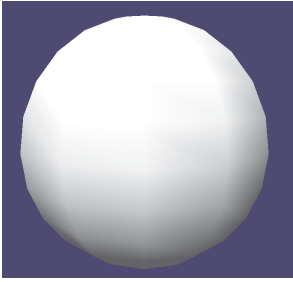
```
MAKE OBJECT SPHERE 3, 40.0
```

would create a sphere with a diameter of 40 units and assign it the ID number 3. However, the sphere produced is constructed from a relatively small number of polygons and hence its curve is not particularly smooth. By using the *rows* and *columns* values, we can control the number of polygons used to construct the sphere and thereby produce a more realistic effect. For example, the line

would create a much smoother sphere. FIG-31.7 shows the difference between the default sphere and the more detailed one.

FIG-31.7

Creating a Smoother
Sphere



The number of polygons
used has greatly increased

CREATE OBJECT SPHERE 3, 40.0

CREATE OBJECT SPHERE 3, 40.0,100,100

However, there's a price to be paid for the more detailed sphere; the more polygons we use when creating any 3D shape, the more work the processor/video card needs to do and this reduces the frames per second that can be achieved.

Activity 31.2

Modify your previous program so that a standard sphere (diameter 10) is created instead of a cube.

Modify the sphere to have 40 columns by 40 rows.

The MAKE OBJECT CYLINDER Statement

A cylinder of a specified height can be created using the MAKE CYLINDER OBJECT statement. The diameter of the cylinder's base automatically matches the height. The statement has the format shown in FIG-31.8.

FIG-31.8

The MAKE OBJECT
CYLINDER Statement



In the diagram:

objno

is the integer value assigned to the cylinder being created.

h

is a real value giving the height and diameter of the cylinder.

For example, we could make a cylinder of height 31.5 units using the statement:

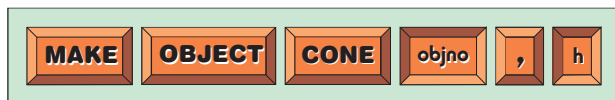
```
MAKE OBJECT CYLINDER 4,31.5
```

The MAKE OBJECT CONE Statement

The MAKE OBJECT CONE statement creates a cone of a specified height. The diameter of the base automatically matches the height. This statement has the format shown in FIG-31.9.

FIG-31.9

The MAKE OBJECT
CONE Statement



In the diagram:

objno

is the integer value assigned to the cone being created.

h

is a real value giving the height of the cone and the diameter of its base.

For example, we could make a cone of height 10.1 units using the statement:

```
MAKE OBJECT CONE 5,10.1
```

Activity 31.3

Modify your previous program to display a cylinder of diameter 5.

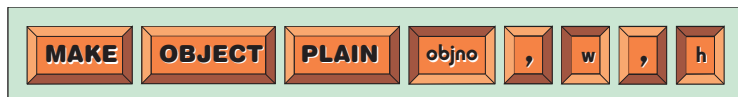
Modify the program again to show a cone of the same height as the cylinder.

The MAKE OBJECT PLAIN Statement

A flat plane standing on the XY plane can be constructed using the MAKE OBJECT PLAIN statement which has the format shown in FIG-31.10.

FIG-31.10

The MAKE OBJECT
PLAIN Statement



Note the spelling used in
the instruction!

In the diagram:

objno

is the integer value assigned to the plane being created.

w

is a real value giving the width of the plane.

h

is a real value giving the height of the plane.

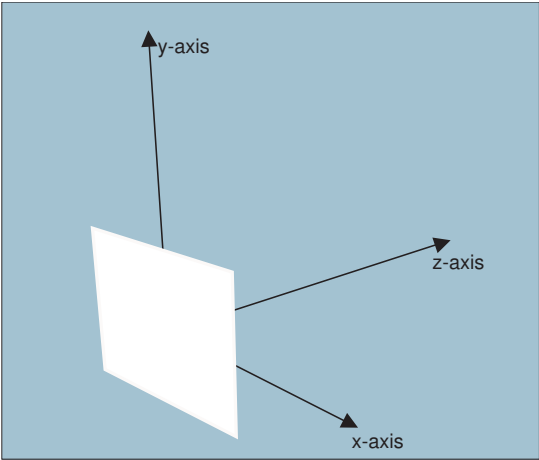
For example, we could create a plane which is 1000 units wide by 500 high using the line:

```
MAKE OBJECT PLAIN 6,1000.0,500.0
```

The centre of the plane will be located at the origin (see FIG-31.11).

FIG-31.11

How a Plane is Positioned
when First Created

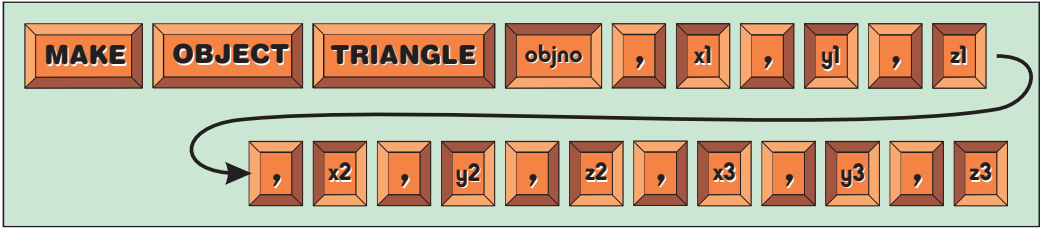


The MAKE OBJECT TRIANGLE Statement

The simplest of all polygons, the triangle, can be constructed using the MAKE OBJECT TRIANGLE statement. The statement requires the positions of all three vertices to be supplied, so this statement contains a significant number of values, as shown in FIG-31.12.

FIG-31.12

The MAKE OBJECT
TRIANGLE Statement



In the diagram:

- objno* is the integer value assigned to the triangle being created.
- x1,y1,z1* are real numbers representing the position of the first vertex.
- x2,y2,z2* are real numbers representing the position of the second vertex.
- x3,y3,z3* are real numbers representing the position of the third vertex.

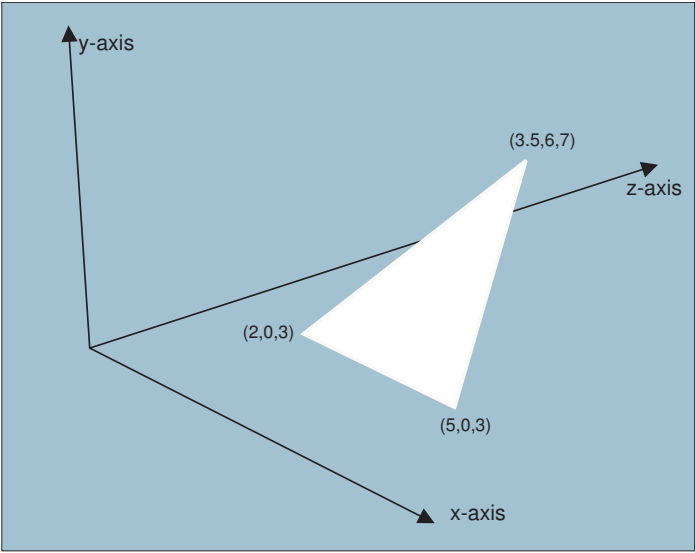
The line

The additional spacing
used within the
instruction is used to
highlight the various
parameter groupings.

```
MAKE OBJECT TRIANGLE 7, 2,0,3, 5,0,3, 3.5,6,7
```

would create the triangle shown in FIG-31.13.

FIG-31.13
Creating a Triangle Object



Notice that, unlike any of the other objects, a triangle can be placed anywhere in 3D space.

Positioning an Object

Other than the triangle, every object is created with its centre at the point (0,0,0). However, once an object has been created, DarkBASIC Pro offers several ways of moving an object to another position.

The POSITION OBJECT Statement

One way to move an object is to use the POSITION OBJECT statement. The object is moved so that its centre is at the position specified. The statement has the format shown in FIG-31.14.

FIG-31.14
The POSITION
OBJECT Statement



In the diagram:

objno is the integer value previously assigned to the object.

x,y,z are real values representing the position to which the object is to be moved. It is the centre of the object that is placed at this position.

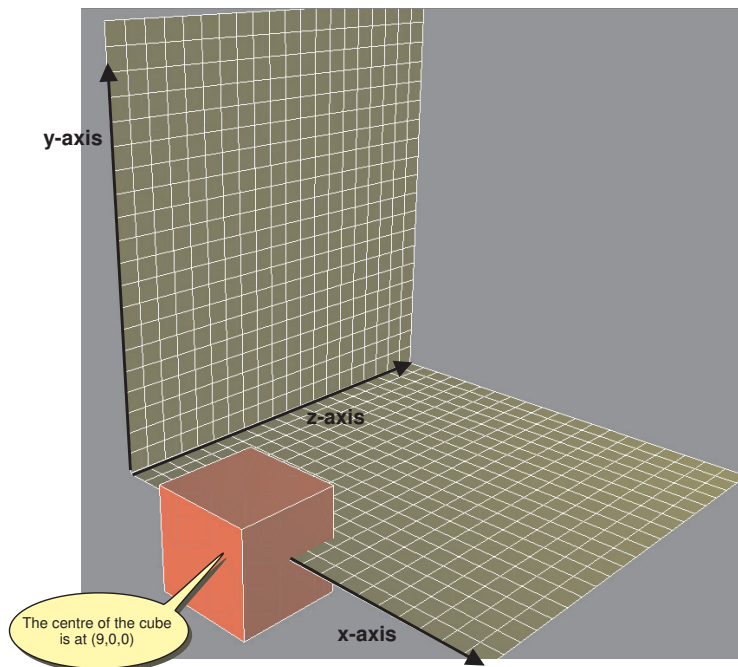
For example, if we wanted the centre of the cube we had created previously to be moved to position (9,0,0), then we would use the statement:

```
POSITION OBJECT 1,9,0,0
```

The result of executing this statement is shown in FIG-31.15.

FIG-31.15

The Result of Moving the
Cube to (9,0,0)



LISTING-31.2

Moving the Cube

LISTING-31.2 is a modification of the previous listing which moves the cube to position (9,0,0) after the user presses a key. The new lines have been highlighted.

```
REM *** Set display resolution and backdrop ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP 0
BACKDROP ON

REM *** Make the cube **
MAKE OBJECT CUBE 1, 10

REM *** Move cube to (9,0,0) after key press ***
WAIT KEY
POSITION OBJECT 1,9,0,0

REM *** End program ***
WAIT KEY
END
```

Activity 31.4

Modify your previous program to match that given in LISTING-31.2.

Add the lines

```
REM *** Move the cube backwards ***
WAIT KEY
POSITION OBJECT 1, 9,0,30
```

so that the object is moved for a second time.

Notice that, in its final position, the cube looks smaller since it has now moved further away from our viewing position.

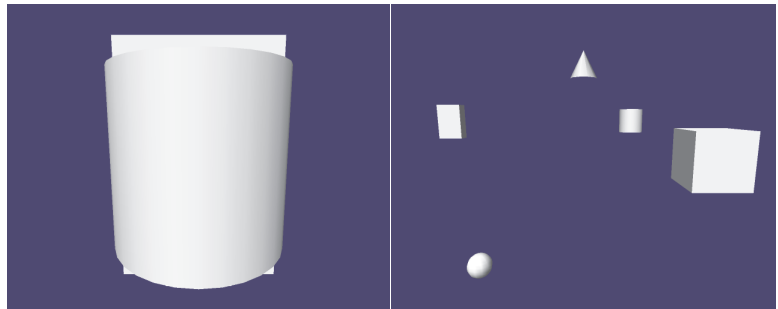
Activity 31.5

Write a program (*object3D02.dbpro*) to create the following objects, and then position each as specified:

Object Number	Object Type	Dimensions	Final Position
1	CUBE	4	9,0,0
2	BOX	10,15,5	-60,0,100
3	SPHERE	7	-30,-40,50
4	CYLINDER	12	25,0,120
5	CONE	12	0,25,100

Add a WAIT KEY statement between each move.

The screen should appear as shown below.



Screen at Start

Screen at End

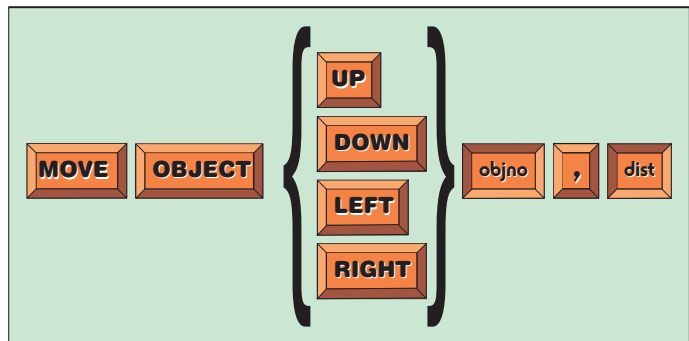
The MOVE OBJECT Statement

The MOVE OBJECT statement can be used to move an object a specified distance from its current location.

There are four possible directions available: RIGHT, LEFT, UP and DOWN. The statement has the format shown in FIG-31.16.

FIG-31.16

The MOVE OBJECT Statement



In the diagram:

UP, DOWN, LEFT, RIGHT

One of these keywords must be used to indicate in which direction the object is to be moved.

objno

is an integer value giving the ID of the object to be moved.

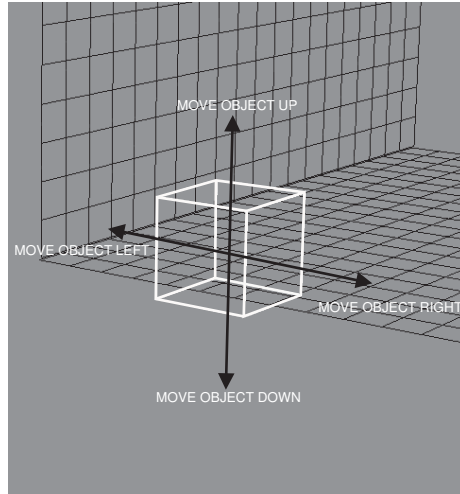
dist

is a real value giving the number of units the object is to be moved.

The direction of movement for each option is shown in FIG-31.17.

FIG-31.17

Using the MOVE
OBJECT Statement



Notice that there is no option to move the object backwards or forwards (i.e. along the z-axis). For example, object 1 could be moved 10.5 units to the right using the statement:

```
MOVE OBJECT RIGHT 1, 10.5
```

Activity 31.6

Create a new program (*object3D03.dbpro*) containing a cube of size 10.

Use POSITION OBJECT to place the cube at (9,0,100).

Now move the cube 31.3 units to the right.

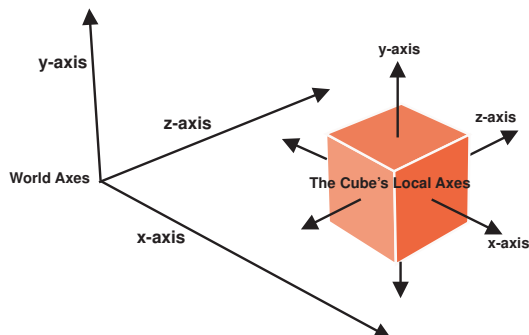
Place a WAIT KEY statement before each action.

Rotating Objects - Absolute Rotation

It is possible to rotate an object about one of its own **local axes**. FIG-31.18 emphasises the difference between the main (or world) axes and local axes.

FIG-31.18

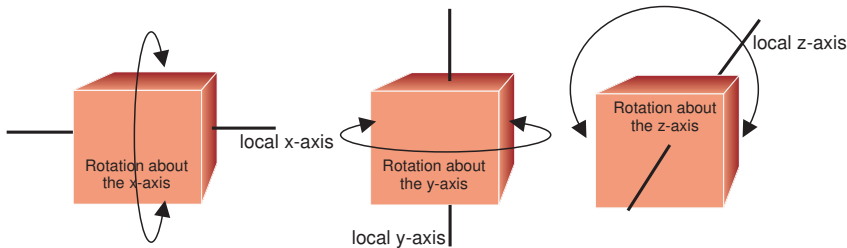
An Object's Local Axes



Possible rotations are shown in FIG-31.19.

FIG-31.19

Possible Rotations about
Local Axes

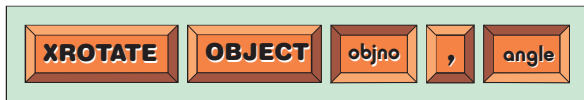


The XROTATE OBJECT Statement

This command causes an identified object to rotate to a specific angle about the object's local x-axis. Rotation is towards the viewer. The statement has the format shown in FIG-31.20.

FIG-31.20

The XROTATE
OBJECT Statement



In the diagram:

objno

is the integer value specifying the object.

angle

is a real number giving the angle (in degrees) to which the object is to be rotated.

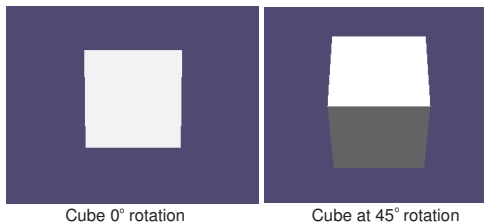
For example, a cube (object 1) can be rotated about the x-axis to 45° using the line

```
XROTATE OBJECT 1, 45.0
```

FIG-31.21 shows the cube before and after rotation.

FIG-31.21

The Effect of the
XROTATE OBJECT
Statement



Activity 31.7

Write a program (*object3D04.dbpro*) which implements the following logic:

```
Set screen resolution to 1280 by 1024
Create a cube (40 units in size)
Move the cube to (0,0,100)
FOR degree := 1 TO 360 DO
    Rotate cube to degree° around the x-axis.
    Wait 1 millisecond
ENDFOR
```

Run the program and check that it performs as expected.

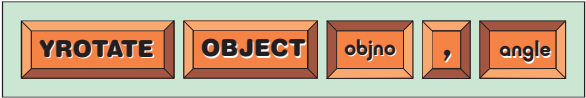
Modify the program so that the cube revolves in the opposite direction about the x-axis.

The YROTATE OBJECT Statement

To rotate an object about its local y-axis we use the YROTATE OBJECT statement which has the format shown in FIG-31.22.

FIG-31.22

The YROTATE OBJECT Statement



In the diagram:

objno is the integer value previously assigned to the object.

angle is a real number giving the angle (in degrees) to which the object is to be rotated.

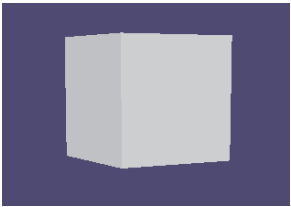
For example, a cube (ID 1) could be rotated about its y-axis to 60° using the line:

```
YROTATE OBJECT 1, 60.0
```

FIG-31.23 shows the cube after a rotation to 60°.

FIG-31.23

The Effect of the YROTATE OBJECT Statement

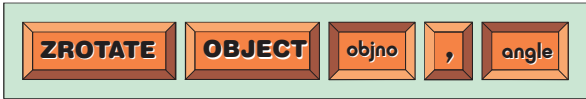


The ZROTATE OBJECT Statement

To rotate an object about its z-axis, we use the ZROTATE OBJECT statement which has the format shown in FIG-31.24.

FIG-31.24

The ZROTATE OBJECT Statement



In the diagram:

objno is the integer value previously assigned to the object.

angle is a real number giving the angle (in degrees) to which the object is to be rotated.

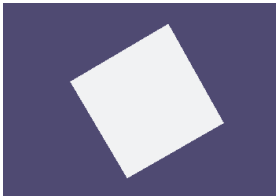
For example, a cube (ID 1) could be rotated about the z-axis to 110° using the line:

```
ZROTATE OBJECT 1, 110.0
```

FIG-31.25 shows the cube after a rotation to 110°.

FIG-31.25

The Effect of the ZROTATE OBJECT Statement



The program in LISTING-31.3 revolves a cube about all three axes at the same time.

LISTING-31.3

Rotating an Object about
all Three Axes

```
REM ** Set display mode ***
SET DISPLAY MODE 1280,1024,32

REM *** Create and position cube ***
MAKE OBJECT CUBE 1, 40
POSITION OBJECT 1, 0, 0, 200

REM *** Rotate cube 1 degree at a time ***
REM *** around all three axes ***
FOR angle = 1 TO 360
  XROTATE OBJECT 1, angle
  YROTATE OBJECT 1, angle
  ZROTATE OBJECT 1, angle
  WAIT 10
NEXT angle

REM *** End program ***
WAIT KEY
END
```

Activity 31.8

Type in and test the program given above (*object3D05.dbpro*).

The ROTATE OBJECT Statement

Rather than use three separate statements to rotate an object about all three axes, the same effect can be achieved using the ROTATE OBJECT statement. This statement takes three values specifying, for each axis, the degree of rotation. The format of the statement is shown in FIG-31.26.

FIG-31.26

The ROTATE
OBJECT Statement



In the diagram:

- objno* is the integer value previously assigned to the object.
- xangle* is a real number giving the angle (in degrees) to which the object is to be rotated about its x-axis.
- yangle* is a real number giving the angle (in degrees) to which the object is to be rotated about its y-axis.
- zangle* is a real number giving the angle (in degrees) to which the object is to be rotated about its z-axis.

Activity 31.9

Rewrite the program you created in the previous Activity, replacing the XROTATE, YROTATE and ZROTATE statements with a single ROTATE OBJECT statement, producing the same effect as before.

All statements in the previous section rotate an object to a specific angle, irrespective

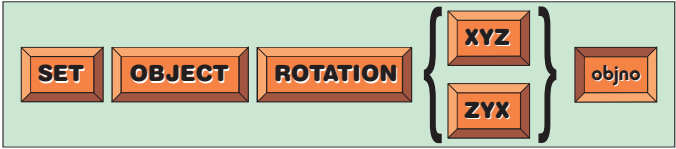
of that object's current inclination. For example, if we use the YROTATE OBJECT statement to turn a cube to 60° , the initial angle of the cube before the statement is executed is irrelevant since the end result will be that the cube will end up at the specified angle of 60° . This type of rotation is known as **absolute rotation**.

The SET OBJECT ROTATION Statement

When an object is rotated about all three axes at the same time, the action is normally implemented by first rotating the object about the x-axis, then the y-axis and finally the z-axis. Of course, it's all done so quickly that the operation will appear to be instantaneous. However, should we want to reverse the order in which the rotations take place (i.e. z-axis first, x-axis last) then we can use the SET OBJECT ROTATION statement. Once set, the order in which the axes are handled will remain on this new setting unless you revert to normal using a second option of the SET OBJECT ROTATION statement, which has the format shown in FIG-31.27.

FIG-31.27

The SET OBJECT
ROTATION ZYX
Statement



In the diagram:

- | | |
|-------|---|
| XYZ | Use this option to return the order of rotations to the default x-axis, y-axis, z-axis order. |
| ZYX | Use this option to set the order of rotations to the z-axis, y-axis, x-axis order. |
| objno | is an integer value specifying the object whose order of rotation is to be modified. |

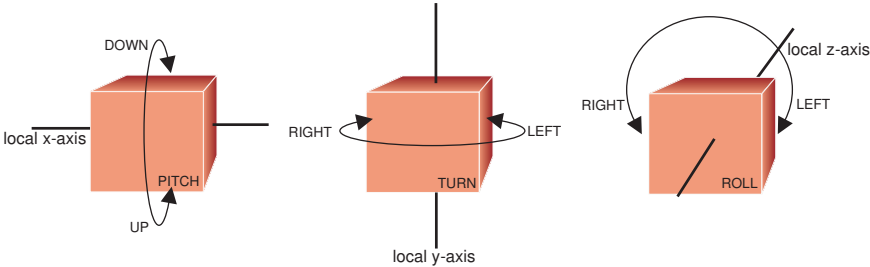
Rotating Objects - Relative Rotation

It is also possible to make an object rotate by a specific angle from its current setting. For example, if a cube has already been rotated 45° about the local y-axis, we can command it to be rotated by a further 60° giving a final rotation position of 105° . This type of rotation - where the angle specified is added to the initial tilt - is known as **relative rotation**.

When using relative rotation, different terms are used for rotation about each axis. Hence, we use the term PITCH for rotation about the x-axis, TURN for rotation about the y-axis and ROLL for rotation about the z-axis (see FIG-31.28).

FIG-31.28

Relative Rotation Terms

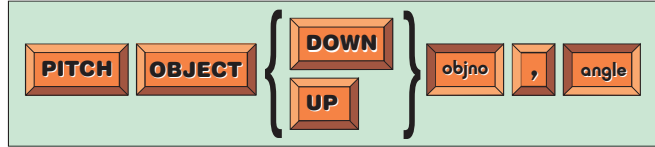


The PITCH OBJECT Statement

We can tilt an object upwards (i.e. rotate it in a positive direction about the x-axis) using the PITCH OBJECT UP statement which has the format shown in FIG-31.29.

FIG-31.29

The PITCH OBJECT Statement



In the diagram:

DOWN, UP

Choose DOWN to make the object rotate clockwise (as viewed from the positive side of the y-axis); choose UP to make the object rotate anticlockwise.

objno

is the integer value previously assigned to the object.

angle

is a real number giving the angle (in degrees) to which the object is to be rotated relative to its current position. The angle can be a positive or negative value.

The program in LISTING-31.4 performs the same function as the one you created in Activity 31.7 where you used the XROTATE statement to rotate a cube through 360°. However, this time the XROTATE statement has been replaced by a PITCH OBJECT UP command.

LISTING-31.4

Using Relative Rotation

```
REM *** Set screen resolution ***
SET DISPLAY MODE 1280,1024,32
REM *** Create and position cube ***
MAKE OBJECT CUBE 1, 40
POSITION OBJECT 1, 0, 0, 200

REM *** Revolve the cube ***
FOR c = 1 TO 360
  PITCH OBJECT UP 1, 1.0
  WAIT 10
NEXT c
REM *** End program ***
WAIT KEY
END
```

Activity 31.10

Type in the program given above (*object3D06.dbpro*) and make sure it is equivalent to the earlier program in Activity 31.7.

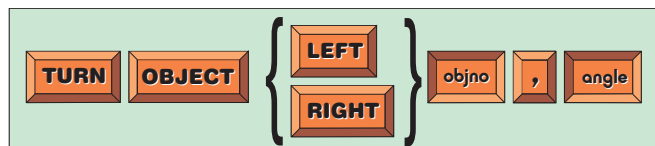
Modify the program so that the cube rotates in the opposite direction.

The TURN OBJECT Statement

This statement allows relative rotation about the y-axis and has the format shown in FIG-31.30.

FIG-31.30

The TURN OBJECT Statement



In the diagram:

LEFT, RIGHT	Choose RIGHT to make the object rotate to the right about the y-axis; choose LEFT to make the object rotate to the left.
<i>objno</i>	is an integer value giving the ID of the object to be rotated.
<i>angle</i>	is a real number giving the angle (in degrees) to which the object is to be rotated relative to its current position. The angle can be a positive or negative value.

Activity 31.11

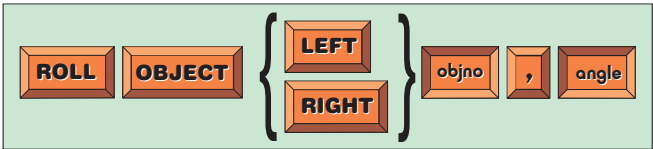
Modify your previous program so that the cube rotates to the right about the y-axis.

The ROLL OBJECT Statement

Relative rotation about the z-axis is achieved using the ROLL OBJECT statement which has the format shown in FIG-31.31.

FIG-31.31

The ROLL OBJECT Statement



In the diagram:

LEFT, RIGHT	Choose RIGHT to make the object rotate to the right about the z-axis; choose LEFT to make the object rotate to the left.
<i>objno</i>	is an integer value giving the ID of the object to be rotated.
<i>angle</i>	is a real number giving the angle (in degrees) to which the object is to be rotated relative to its current position. The angle can be a positive or negative value.

Activity 31.12

Modify your previous program so that the cube rotates to the left about the z-axis.

The POINT OBJECT Statement

The main polygon of a 3D object is directed towards the player's viewpoint when it is created. This polygon can be rotated to face any point in space using the POINT OBJECT statement. This statement has the format shown in FIG-31.32.

FIG-31.32

The POINT OBJECT Statement



In the diagram:

- objno*
- is an integer value giving the ID of the object to be affected.
- x,y,z*
- are the coordinates of the point in space at which the main polygon of the 3D object is to face.

In the program shown in LISTING-31.5 a cube is made to face the point (45,45,0) using the statement:

```
POINT OBJECT 1,45,45,0
```

LISTING-31.5

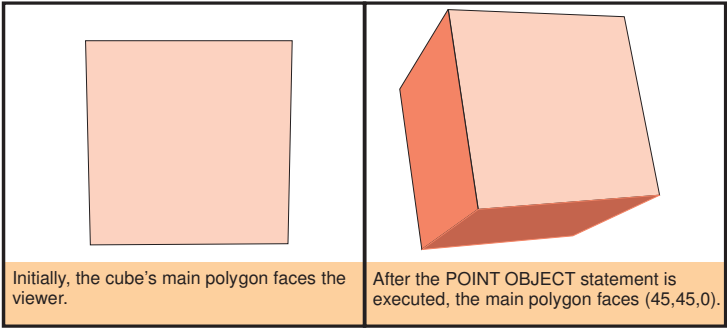
Using the POINT OBJECT Statement

```
REM *** Set display resolution and backdrop ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP 0
BACKDROP ON
REM *** Make the set of objects ***
MAKE OBJECT CUBE 1,40
REM *** Move cube to (0,0,100) after key press ***
WAIT KEY
POSITION OBJECT 1,0,0,100
REM *** point cube at (45,45,0)***
WAIT KEY
POINT OBJECT 1,45,45,0
REM *** End program ***
WAIT KEY
END
```

The result is shown in FIG-31.33.

FIG-31.33

Turning an Object to Face a Specified Point



Activity 31.13

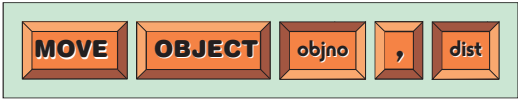
Type in and test the program given in LISTING-31.5 (*object3D07.dbpro*).
Modify the program to make the cube face the point (-20,17,-10).

The MOVE OBJECT *distance* Statement

We've already encountered a MOVE OBJECT statement which allows an object to be moved up, down, left, or right, but a second version of MOVE OBJECT exists which will move an object in the direction its main polygon is facing. This statement has the format shown in FIG-31.34.

FIG-31.34

The MOVE OBJECT
distance Statement



In the diagram:

- objno*
- is an integer value specifying the object to be moved.
- dist*
- is a real value specifying the distance to be moved.

Activity 31.14

In your previous program, immediately after the POINT OBJECT statement, add the following lines:

```
REM *** Move cube ***
WAIT KEY
MOVE OBJECT 1, 20
```

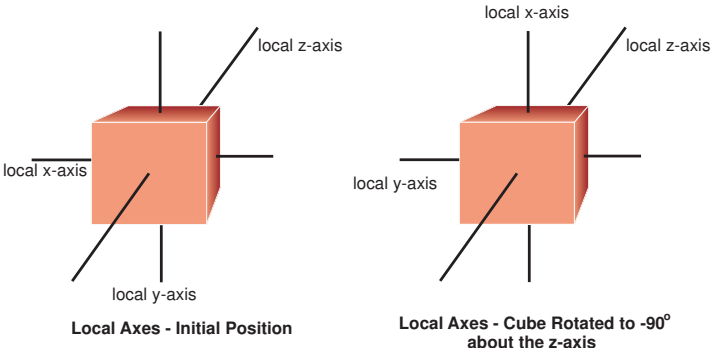
Run the updated program.

The FIX OBJECT PIVOT Statement

When an object rotates, its local axes rotate with it. In FIG-31.35 we see a cube and its local axes before and after it has been rotated to -90° about its local z-axis.

FIG-31.35

How the Local Axes are
Affected When an
Object is Rotated



If we now rotate the cube about its own x-axis, it will turn left-to-right rather than up-and-over, because its x-axis has shifted position. This is demonstrated in LISTING-31.6 where the cube is rotated a full 360° about its x-axis, rotated by -90° about its z-axis and then rotated a full 360° about its x-axis for a second time.

LISTING-31.6

Local Axes Move with
the Object

```
REM *** Set display resolution ***
SET DISPLAY MODE 1280,1024,32
REM *** Make and position cube ***
MAKE OBJECT CUBE 1,40
POSITION OBJECT 1,0,0,100

REM *** Rotate cube 360 about x-axis ***
FOR degree = 0 TO 360
  XROTATE OBJECT 1, degree
  WAIT 1
NEXT degree
```

continued on next page

LISTING-31.6
(continued)

Local Axes Move with
the Object

```
REM *** Rotate cube to -90 about z-axis
FOR degree = 0 TO -90 STEP -1
  ZROTATE OBJECT 1, degree
  WAIT 10
NEXT degree

REM *** Rotate cube 360 about x-axis ***
FOR degree = 0 TO 360
  XROTATE OBJECT 1, degree
  WAIT 1
NEXT degree

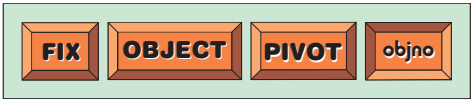
REM *** End program ***
WAIT KEY
END
```

Activity 31.15

Type in and test the program given in LISTING-31.6 (*object3D08.dbpro*).

FIG-31.36

The FIX OBJECT PIVOT
Statement



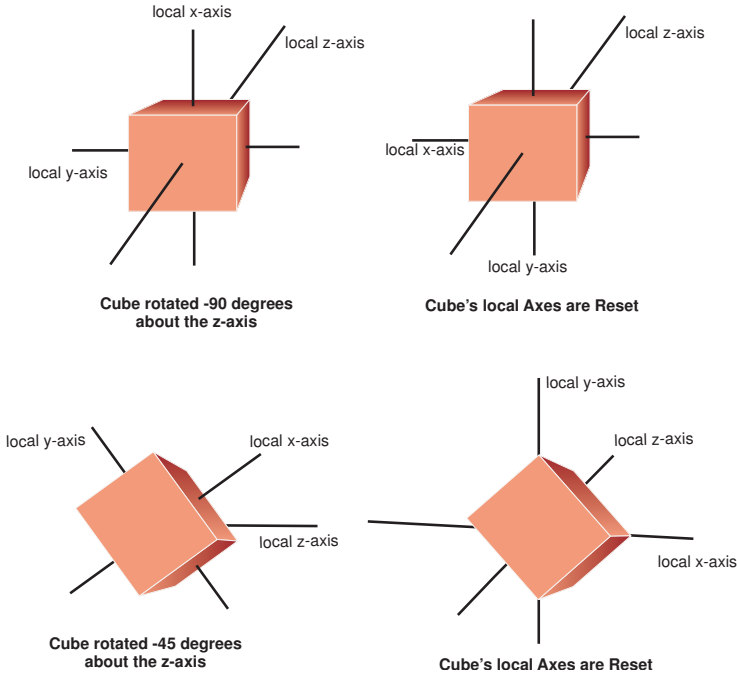
In the diagram:

objno is an integer value specifying the object whose local axes are to be reset.

When this statement is executed, the object in question has its local axes reset so that the x-axis lies left-to-right, the y-axis top-to-bottom, and the z-axis in-to-out (see FIG-31.37).

FIG-31.37

The Effect of Using FIX
OBJECT PIVOT



Activity 31.16

In your previous program (*object3D08.dbpro*), add the line

```
FIX OBJECT PIVOT 1
```

before the final FOR loop structure.

How does this affect the rotation of the cube?

Modify the program again so that the cube is only rotated to -45° in the second FOR loop.

How is the cube's rotation affected this time?

Resizing Objects

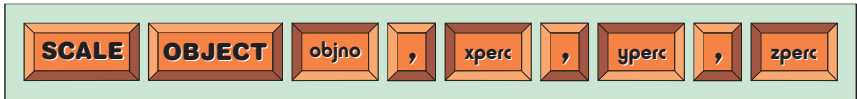
It is possible to change the size of an object after it has been created. You have the option to resize one, two, or all three of the object's dimensions. This allows you to make an object uniformly larger or smaller, or to distort the original shape by changing each dimension by differing amounts.

The SCALE OBJECT Statement

Resizing an existing 3D object is achieved using the SCALE OBJECT statement which has the format shown in FIG-31.38.

FIG-31.38

The SCALE OBJECT Statement



In the diagram:

objno

is the integer value previously assigned to the 3D object.

xperc

is a real number giving the new size of the object's x dimension as a percentage of its original size in that dimension. For example, a value of 100.0 will retain the current size, while 200.0 would double the object's length in the x dimension, and 50.0 would halve it.

yperc

is a real number giving the new size of the object's y dimension as a percentage of its original size in that dimension.

zperc

is a real number giving the new size of the object's z dimension as a percentage of its original size in that dimension.

The program in LISTING-31.7 creates a sphere with a radius of 20 units. The sphere is then resized so that the x dimension is doubled and the z dimension reduced to 10 units. The new shape is then rotated about the y-axis. The user must press ESC to terminate the program.

LISTING-31.7

Resizing an Object

```
REM *** Set display resolution ***
SET DISPLAY MODE 1280,1024,32

REM *** Make and position the sphere **
MAKE OBJECT SPHERE 1, 20
POSITION OBJECT 1, 0,0,200

REM *** Resize sphere ***
SCALE OBJECT 1, 200.0,100.0,50.0

REM *** Rotate shape ***
DO
    TURN OBJECT RIGHT 1, 1.0
LOOP

REM *** End program ***
WAIT KEY
END
```

Activity 31.17

Type in and test the program in LISTING-31.6 (*object3D09.dbpro*).

Modify the program so that a cone is used in place of the sphere.

Showing and Hiding Objects

Any 3D object is immediately visible from the moment it is created (assuming it's within view), but it is possible to hide an object using the HIDE OBJECT statement, making it reappear later using the SHOW OBJECT command.

The HIDE OBJECT Statement

An object can be made invisible using the HIDE OBJECT statement which has the format shown in FIG-31.39.

FIG-31.39

The HIDE OBJECT
Statement



In the diagram:

objno

is the integer value previously assigned to the 3D object which is to be hidden.

The SHOW OBJECT Statement

An object which has been previously hidden can be made to reappear using the SHOW OBJECT statement which has the format shown in FIG-31.40.

FIG-31.40

The SHOW OBJECT
Statement



In the diagram:

objno

is the integer value previously assigned to the hidden 3D object which is to reappear.

The program in LISTING-31.8 rotates a cube continually, hiding the cube when 'h' is pressed and showing it again when 's' is pressed.

LISTING-31.8

Hiding and Showing
Objects

```
REM *** Set display resolution ***
SET DISPLAY MODE 1280,1024,32

REM *** Make and position cube ***
MAKE OBJECT CUBE 1, 20
POSITION OBJECT 1, 0,0,200

REM *** Rotate object ***
DO
  PITCH OBJECT UP 1, 1.0
  REM *** Read key ***
  ch$ = INKEY$()
  REM *** IF its s - show cube ***
  IF ch$ = "s"
    SHOW OBJECT 1
  ENDIF
  REM *** IF its h - hide cube ***
  IF ch$ = "h"
    HIDE OBJECT 1
  ENDIF
LOOP
REM *** End program ***
END
```

Activity 31.18

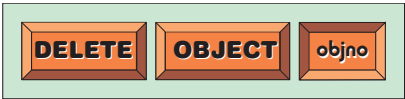
Type in and test the program given above (*object3D10.dbpro*).

The DELETE OBJECT Statement

When a 3D object is no longer required, its RAM space can be released using the DELETE OBJECT statement which has the format shown in FIG-31.41.

FIG-31.41

The DELETE OBJECT
Statement



In the diagram:

objno

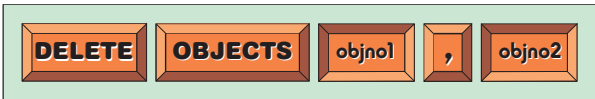
is an integer value specifying the ID of the 3D object to be deleted.

The DELETE OBJECTS Statement

If we need to delete several objects at one time, then the most efficient way to do this is to use the DELETE OBJECTS statement which has the format shown in FIG-31.42.

FIG-31.42

The DELETE
OBJECTS Statement



In the diagram:

objno1

is an integer value specifying the lowest ID of the 3D objects to be deleted.

objno2

is an integer value specifying the highest ID of the 3D objects to be deleted.

For example, if we needed to delete 10 3D objects with ID values ranging from 8 to 17, then we would use the statement:

```
DELETE OBJECTS 8,17
```

Copying a 3D Object

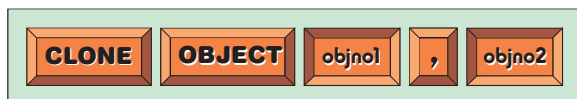
We can create a copy of an existing 3D object in one of two ways, as described below.

The CLONE OBJECT Statement

The CLONE OBJECT statement creates an independent copy of an existing 3D object. The statement has the format shown in FIG-31.43.

FIG-31.43

The CLONE OBJECT Statement



In the diagram:

objno1

is an integer value specifying the ID to be assigned to the object being created.

objno2

is an integer value specifying the ID of the existing object to be copied.

The program in LISTING-31.9 uses the CLONE OBJECT statement to create a duplicate cube.

LISTING-31.9

Creating a Copy of a 3D Object

```
REM *** Set display resolution ***
SET DISPLAY MODE 1280,1024,32
REM *** Make and position cube ***
MAKE OBJECT CUBE 1,40
POSITION OBJECT 1,-40,0,100
REM *** Rotate cube ***
WAIT KEY
XROTATE OBJECT 1 , -45
REM *** Copy cube ***
WAIT KEY
CLONE OBJECT 2,1
REM *** Position copy ***
POSITION OBJECT 2, 40,0,50
REM *** Delete original cube ***
WAIT KEY
DELETE OBJECT 1
REM *** End program ***
WAIT KEY
END
```

Activity 31.19

Type in and test the program given in LISTING-31.9 (*object3D11.dbpro*).

Is the copied cube rotated to the same angle as the original?

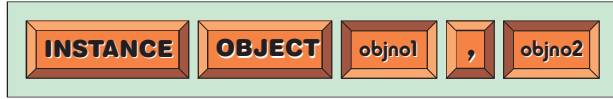
Is the copied cube deleted when the original cube is removed?

The INSTANCE OBJECT Statement

A second way to duplicate an existing object is to use the INSTANCE OBJECT statement. This statement has the format shown in FIG-31.44.

FIG-31.44

The INSTANCE
OBJECT Statement



In the diagram:

objno1

is an integer value specifying the ID to be assigned to the object being created.

objno2

is an integer value specifying the ID of the existing object to be copied.

Although this may seem to have the same affect as the CLONE OBJECT statement, in fact the two statements differ in how data about the copied object is held. When CLONE OBJECT is used, the new object has its own independent data area; with INSTANCE OBJECT the two objects share parts of the same data area. The consequence of this is that objects created using INSTANCE OBJECT will disappear if the original object from which they were created is deleted.

Activity 31.20

Modify your last program, replacing the CLONE OBJECT statement with a INSTANCE OBJECT statement.

How does this change affect the operation of the program?

Change the DELETE OBJECT statement so that object 2, rather than object 1, is deleted. How does this affect the program?

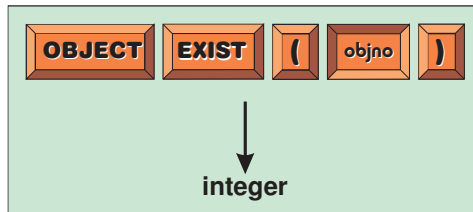
Retrieving Data on 3D Objects

The OBJECT EXIST Statement

We can check that an object of a specified ID actually exists using the OBJECT EXIST statement which has the format shown in FIG-31.45.

FIG-31.45

The OBJECT EXIST
Statement



In the diagram:

objno

is an integer specifying the ID of the object to be checked.

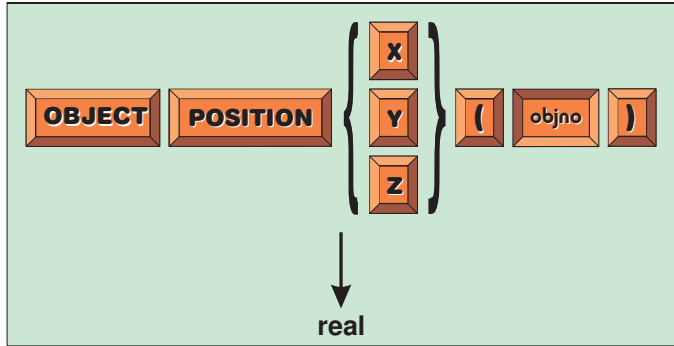
If the object exists, 1 is returned, otherwise zero is returned.

The OBJECT POSITION Statement

The exact position of a 3D object's centre can be determined using the OBJECT POSITION statement. Three variations of the statement exist, with each variation returning one of the object's ordinates. The statement has the format shown in FIG-31.46.

FIG-31.46

The OBJECT POSITION
Statement



In the diagram:

X,Y,Z

One of these options should be chosen. Choose X if the x-ordinate of the specified object is required, Y for the y-ordinate, and Z for the z-ordinate.

objno

is an integer value specifying the object whose ordinate is to be returned.

For example, we could determine the position in space of object 1's centre using the lines:

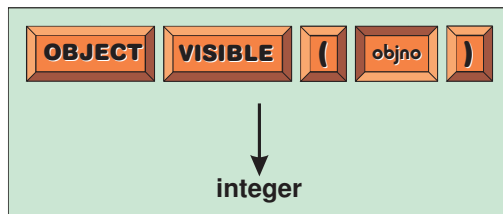
```
x = OBJECT POSITION X(1)
y = OBJECT POSITION Y(1)
z = OBJECT POSITION Z(1)
PRINT "Object 1 has its centre at (" ,x," ",y," ",z," )"
```

The OBJECT VISIBLE Statement

The OBJECT VISIBLE statement returns 1 if a specified 3D object is currently, visible; if the object is hidden, the value zero is returned. The statement has the format shown in FIG-31.47.

FIG-31.47

The OBJECT VISIBLE
Statement



In the diagram:

objno

is an integer value specifying the ID of the object to be checked.

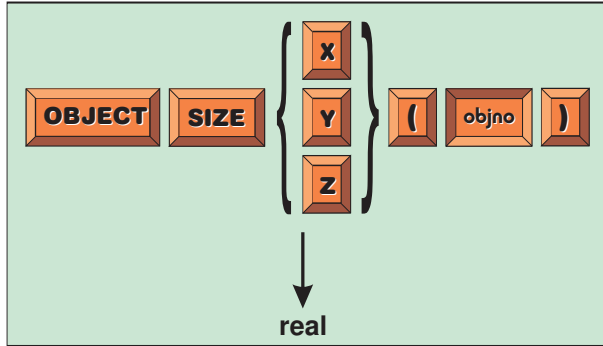
If the object is currently showing, 1 is returned, otherwise zero is returned.

The OBJECT SIZE Statement

The dimensions of a specified object can be determined using the OBJECT SIZE statement. Like OBJECT POSITION, there are three variations available in this statement, each returning one dimension of the object in question. The format for this statement is given in FIG-31.48.

FIG-31.48

The OBJECT SIZE
Statement



In the diagram:

X,Y,Z

One of these options should be chosen. Choose X if the width of the specified object is required, Y for the height, and Z for the depth.

All three options can be omitted and the statement will return an overall value for the size of the 3D object.

objno

is an integer value specifying the object whose dimension is to be returned.

The value returned by the statement is real and, because of rounding errors, this may be slightly out. For example, if we create a cube (object 1) 40 units in all directions, then the statement

```
PRINT "Width ", OBJECT SIZE X(1)
```

will display the value 39.9999961853.

Also, the OBJECT SIZE (1) statement - with no reference to any specific dimension - gives an overall size based on all three dimensions.

Activity 31.21

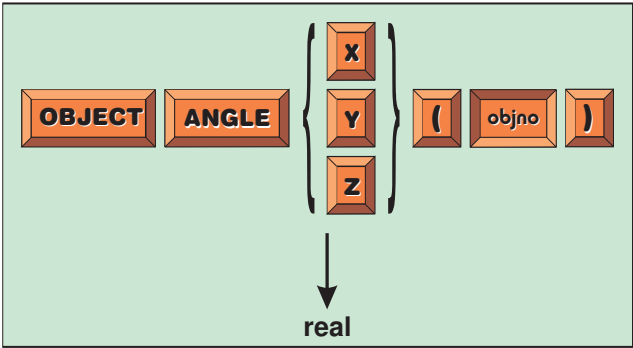
Write a program (*object3D12.dbpro*) which creates a box of random size (using limits 5 to 50) and then displays the box's width, height and depth.

The OBJECT ANGLE Statement

The angle to which an object has been rotated about any of its local axes can be determined using the OBJECT ANGLE statement which has the format shown in FIG-31.49.

FIG-31.49

The OBJECT ANGLE
Statement



In the diagram:

X,Y,Z

One of these options should be chosen. Choose X if the rotation about the local x axis is required, Y for rotation about the y-axis, and Z for the rotation about the z-axis.

objno

is an integer value specifying the object whose rotation angle is to be returned.

Activity 31.22

Modify your previous program so that the box object is rotated by a random number of degrees about all three axes. Display the amount of rotation in each case.

Controlling an Object's Rotation Using the Mouse

In the example that follows, we're going to make a cube face towards the mouse pointer. As the user moves the mouse pointer about the screen, so the cube will continually re-orientate itself to face the pointer.

Before looking at the code, we have one main obstacle to overcome. The mouse pointer commands (MOUSE X()) and MOUSE Y()) use a 2D coordinate system with the origin at the top left corner of the screen; 3D objects use a coordinate system in which the origin is (initially, at least) at the centre of the screen. To convert the mouse's x ordinate readings to 3D space we need to use the line:

$$x3D = \text{MOUSE } X() - \text{SCREEN WIDTH}() / 2$$

The y ordinate also needs to have it's sign changed, since for the mouse the positive section of the y-axis is down, while in 3D space the positive section of the y-axis is up! We can solve this with the line:

$$y3D = -(\text{MOUSE } Y() - \text{SCREEN HEIGHT}() / 2)$$

Of course, there is no third dimension as far as the mouse pointer is concerned, so we'll keep that set to zero.

We're now ready to describe the logic required by the program:

Create cube
Move cube backwards to reduce its apparent size
DO

Get mouse coordinates and convert to 3D space
 Make cube point at these coordinates
 LOOP

The code for the program is given in LISTING-31.10.

LISTING-31.10

Making a 3D Object Face
 the Mouse Pointer

```
REM *** Set display resolution ***
SET DISPLAY MODE 1280,1024,32
REM *** Make and position cube ***
MAKE OBJECT CUBE 1, 40
POSITION OBJECT 1,0,0,100
DO
  REM *** Convert mouse 2D coords to 3D ***
  x3D = MOUSEX() - SCREEN WIDTH() / 2
  y3D = -(MOUSEY() - SCREEN HEIGHT() / 2)
  REM *** Re-orient cube ***
  POINT OBJECT 1,x3D,y3D,0
LOOP
REM *** End program ***
END
```

Activity 31.23

Type in and test the code given above (*object3D13.dbpro*).

Modify the program to use two spheres, set side-by-side, both of which should face towards the mouse pointer.

Wireframe and Culling

The SET OBJECT WIREFRAME Statement

It is possible to show a 3D object in wireframe mode (which show only the edges of the polygons that make up a shape) using the SET OBJECT WIREFRAME statement which has the format shown in FIG-31.50.

FIG-31.50

The SET OBJECT
 WIREFRAME Statement



In the diagram:

objno

is an integer value identifying the object which is to have its display mode altered.

mode

is 0 or 1.

0 - solid mode

1 - wireframe mode

LISTING-31.11 demonstrates the use of this statement, switching between solid and wireframe mode every time a key is pressed.

LISTING-31.11

Switching Between
 Normal and Wireframe
 Mode

```
REM *** Set screen mode ***
SET DISPLAY MODE 1280,1024,32
REM *** Make and position cube ***
MAKE OBJECT CUBE 1, 40
POSITION OBJECT 1,25,0,100
REM *** Start in solid mode ***
wire = 0
```

continued on next page

LISTING-31.11

(continued)

Switching Between
Normal and Wireframe
Mode

```
REM *** Rotate cube ***
DO
REM *** IF key pressed, switch mode ***
IF INKEY$() <> ""
    wire = 1 - wire
    SET OBJECT WIREFRAME 1, wire
ENDIF
PITCH OBJECT DOWN 1, 1.0
TURN OBJECT LEFT 1, 1.0
LOOP
REM *** End program ***
END
```

Activity 31.24

Type in and test the program in LISTING-31.11 (*object3D14.dbpro*).

The SET OBJECT CULL Statement

Under normal circumstances, it is impossible to see every part of a 3D object at the same time. The polygons that make up the hidden parts of an object are not drawn by the computer. Obviously, this saves processing time and creates a more realistic effect. If we take a second look at the previous program when running in wireframe mode, we'll see that the polygons at the back of the cube are not drawn. This elimination of hidden polygons is known as **culling**. Culling can be toggled on or off using the SET OBJECT CULL statement which has the format shown in FIG-31.51.

FIG-31.51

The SET OBJECT
CULL Statement



In the diagram:

objno

is an integer value identifying the object which is to have its cull mode altered.

mode

is 0 or 1.

0 - culling off

1 - culling on

Activity 31.25

Modify your last program by adding the line

```
SET OBJECT CULL 1,0
```

immediately before the DO..LOOP structure. *Notice that the hidden polygons are now being drawn.*

Modify the program again so that pressing the *w* key toggles between wireframe and solid mode and that pressing *c* toggles between culling on and culling off.

There's a slight problem when it comes to displaying cylinders and cones, as we can see from the output produced by LISTING-31.12.

LISTING-31.12

A Problem with Cones and Cylinders

```
REM *** Set screen mode ***
SET DISPLAY MODE 1280,1024,32
REM *** Make and position cone and cylinder ***
MAKE OBJECT CONE 1, 5
POSITION OBJECT 1,-6,-5,0
XROTATE OBJECT 1, 45
MAKE OBJECT CYLINDER 2,5
POSITION OBJECT 2, 6,-5,0
REM *** End program ***
WAIT KEY
END
```

Activity 31.26

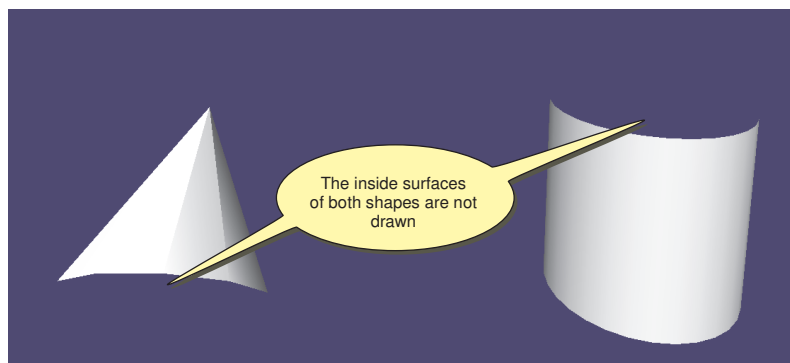
Type in and test the program in LISTING-31.12 (*object3D15.dbpro*).

What problem occurs with both shapes?

The inside surfaces of both shapes have not been drawn (see FIG-31.52).

FIG-31.52

Hidden Surfaces on
Cones and Cylinders



But we can solve this problem by switching off culling, so that the hidden polygons are drawn.

Activity 31.27

Modify your previous program so that culling is switched off for both the cone and the cylinder.

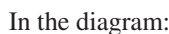
Storage Methods

When a 3D object is shown on screen, the coordinates of its vertices are stored in memory in an area known as a **vertex buffer**. Normally, each object will have its own vertex buffer. However, some video cards allow different objects to share vertex buffers, other video cards don't. As a default, 3D objects in DARKBASIC Pro do not share vertex buffers - this ensures compatibility with the maximum number of video cards. However, it is possible to force vertex buffer sharing and, if your video card can handle this, improve the performance of your program.

The SET GLOBAL OBJECT CREATION Statement

To force vertex buffer sharing, we use the SET GLOBAL OBJECT CREATION statement which has the format shown in FIG-31.53.

The SET GLOBAL OBJECT CREATION Statement



0 or 1. 0 - no vertex buffer sharing (this is the default setting); 1 - vertex buffer sharing allowed.

Even if your own video card does allow vertex buffer sharing, your customer's may not, so it's probably best to ignore this option.

Summary

- A point in 3D space is specified using x, y, and z coordinates.
- There are three main planes in 3D space - XY, YZ, and XZ.
- The computer screen uses a positive-up, negative-down y-axis when operating in 3D mode. This is the opposite from the 2D settings.
- The positive z-axis travels away from the viewer "into" the screen.
- A point in space is known as a vertex.
- A set of vertices, when joined, form a polygon.
- The join between two vertices is known as an edge.
- Basic 3D shapes are known as primitives.
- When first created a 3D object has its centre position at the origin.
- The x and y axes intersect at the centre of the screen at start-up.
- Use the MAKE OBJECT CUBE to create a cube.
- Use MAKE OBJECT BOX to create a cuboid.
- Use MAKE OBJECT SPHERE to create a sphere.
- The number of polygons used can be specified when creating a sphere.
- Use MAKE OBJECT CYLINDER to create a cylinder.
- Use MAKE OBJECT CONE to create a cone.
- Use MAKE OBJECT PLANE to create a plane.
- A plane is initially oriented as an XY plane.
- Use MAKE OBJECT TRIANGLE to create a triangle.
- The initial position of a triangle is determined by the vertices given.
- Use POSITION OBJECT to place the centre of an object at a new location.

- Use MOVE OBJECT to move an object along the x or y axis.
- Every 3D object has its own local axes with the origin at the centre of the object.
- Use the XROTATE, YROTATE or ZROTATE OBJECT statements to rotate an object to a specific angle about one of its local axes.
- Use ROTATE OBJECT to rotate an object to specific angles about all three local axes at the same time.
- Use PITCH, TURN or ROLL OBJECT statements to rotate an object by a number of degrees around a given axis.
- Use POINT OBJECT to make an object face towards a specified point.
- Use MOVE OBJECT *distance* to move the object a specified number of units in the direction in which an object is pointing.
- Use FIX OBJECT PIVOT to reset an object's local axes to be in line with the global axes.
- Use SCALE OBJECT to change the dimensions of an object.
- Use HIDE OBJECT to make an object invisible.
- Use SHOW OBJECT to make an invisible object reappear.
- Use DELETE OBJECT to erase an object from RAM.
- Use DELETE OBJECTS to erase a group of objects from RAM.
- Use CLONE OBJECT to make an independent copy of an existing object.
- Use INSTANCE OBJECT to create a dependent copy of an existing object.
- Use OBJECT EXIST to check if a specified object exists.
- Use OBJECT VISIBLE to check if a specified object is visible.
- Use OBJECT POSITION to determine the position in space of an object's centre.
- Use OBJECT SIZE to determine the dimensions of a specified object.
- Use OBJECT ANGLE to determine the current angle of rotation a specified object has about its local axes.
- Use SET OBJECT WIREFRAME to display a 3D object in wireframe or normal mode.
- Use SET OBJECT CULL to toggle culling for a specified 3D object.
- Use SET GLOBAL OBJECT CREATION to enable/disable vertex buffer sharing.

Merging Primitives

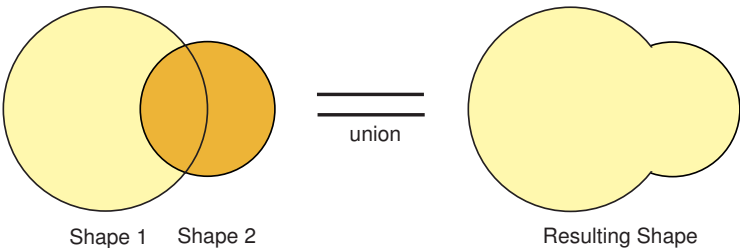
Introduction

DarkBASIC Pro version 1.058 introduced statements which allow us to create new shapes by merging two primitives. There are three basic options available:

Create the new shape from the combination of the two original shapes - known as **union** (see FIG-31.54).

FIG-31.54

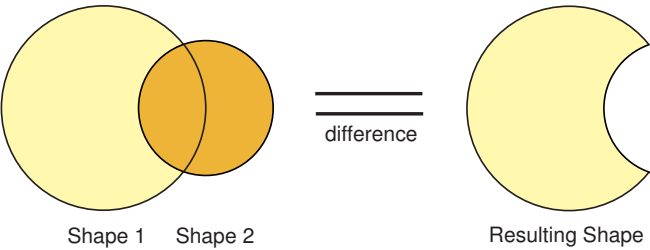
Shape Union



Create the new shape by removing the overlapping section of shape 2 from shape 1 - known as **difference** (see FIG-31.55).

FIG-31.55

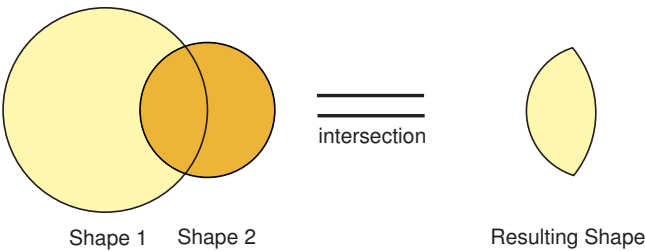
Shape Difference



Create the new shape from the overlapping area between shape 1 and shape 2 - known as **intersection** (see FIG-56).

FIG-31.56

Shape Intersection



The Statements

When we join two shapes using the merge statements, the resulting shape is stored in a format known as **Constructive Solid Geometry (CSG)**. We need not concern ourselves with the details of this format, and it is only mentioned here so that you know the meaning of the initials used in the statements.

The PERFORM CSG UNION Statement

We can create a new shape from the union of two existing shapes using the PERFORM CSG UNION statement which has the format shown in FIG-31.57.

FIG-31.57

The **PERFORM CSG UNION** Statement



In the diagram:

- objno1* is an integer value specifying the ID of the first 3D object to be used in the union.
- objno2* is an integer value specifying the ID of the second 3D object to be used in the union.

The statement modifies the shape of *objno1* without affecting *objno2*. Normally, the programmer would delete the second object once the union is completed.

The program in LISTING-31.13 demonstrates the union of a cube and a box. After the box has been deleted, the resulting shape is then rotated about its local x and y axes.

LISTING-31.13

Creating a New 3D Shape using Union

This program positions the camera which is responsible for the picture we see on the screen. Full details of camera usage are covered in Chapter 33.

```
REM *** Set screen resolution and background ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP 0
BACKDROP ON

REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 0,0,-100

REM *** Create two shapes used ***
MAKE OBJECT CUBE 1,40
MAKE OBJECT BOX 2,10,30,10
POSITION OBJECT 2,0,15,0

REM *** Let viewer see position of shapes ***
WAIT KEY

REM *** Union shapes ***
PERFORM CSG UNION 1,2

REM *** Remove object 2 ***
DELETE OBJECT 2

REM *** Rotate new shape ***
DO
    TURN OBJECT LEFT 1,1.0
    PITCH OBJECT UP 1,1.0
LOOP

REM *** End program ***
END
```

Activity 31.28

Type in and test the program given in LISTING-31.13 (*object3D16.dbpro*).

Try changing the second object to a sphere of diameter 10 and change the **POSITION OBJECT** statement to read

```
POSITION OBJECT 2, 0, 25, 0
```

As you've just discovered, the UNION statement only works predictably with cubes and boxes. Other shapes give unpredictable results (although the cone is close).

The PERFORM CSG DIFFERENCE Statement

The PERFORM CSG DIFFERENCE statement removes from object 1 the volume it shares in common with object 2. The statement has the format shown in FIG-31.58.

FIG-31.58

The PERFORM CSG
DIFFERENCE Statement



In the diagram:

objno1 is an integer value specifying the ID of the first 3D object to be used in the difference operation.

objno2 is an integer value specifying the ID of the second 3D object to be used in the difference operation.

As before, the second object is unaffected by the operation and would normally be deleted. Also, we are again restricted to cubes and boxes if we are to obtain consistent results.

Activity 31.29

Restore your last project to its original code (as shown in LISTING-31.13).

Change the union operation to a difference operation and observe the new shape created.

Modify the width and depth of the box to be 35.

The PERFORM CSG INTERSECTION Statement

The PERFORM CSG INTERSECTION statement does not perform intersection as defined at the beginning of this chapter, but it does create a different shape from that produced by the PERFORM CSG DIFFERENCE statement and so is worth looking at. The statement has the format shown in FIG-31.59.

FIG-31.59

The PERFORM CSG
INTERSECTION
Statement



In the diagram:

objno1 is an integer value specifying the ID of the first 3D object to be used in the intersection operation.

objno2 is an integer value specifying the ID of the second 3D object to be used in the intersection operation.

Again, the second object would normally be deleted and we are restricted to cubes and boxes if we are to obtain predictable results.

Activity 31.30

Modify your previous program and determine how the shape created by the `PERFORM CSG INTERSECTION` differs from that produced by `PERFORM CSG DIFFERENCE`.

Summary

- Cubes and boxes can be merged to create new shapes.
- Use `PERFORM CSG UNION` to modify a shape so that it becomes the amalgamation of the original two shapes.
- Use `PERFORM CSG DIFFERENCE` to modify a shape so that the volume it shares with a second shape is removed.
- Use `PERFORM CSG INTERSECTION` to modify a shape to remove polygons which touch the second shape.

Solutions

Activity 31.1

No solution required.

Activity 31.2

```
1.
REM *** Set display and backdrop ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP 0
BACKDROP ON
REM *** Make the sphere ***
MAKE OBJECT SPHERE 1,10
REM *** End program ***
WAIT KEY
END

2.
REM *** Set display and backdrop ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP 0
BACKDROP ON
REM *** Make the sphere ***
MAKE OBJECT SPHERE 1,10,40,40
REM *** End program ***
WAIT KEY
END
```

Activity 31.3

```
1.
REM *** Set display and backdrop ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP 0
BACKDROP ON
REM *** Make the cylinder ***
MAKE OBJECT CYLINDER 1,5
REM *** End program ***
WAIT KEY
END

2.
REM *** Set display and backdrop ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP 0
BACKDROP ON
REM *** Make the cone ***
MAKE OBJECT CONE 1,5
REM *** End program ***
WAIT KEY
END
```

Activity 31.4

```
REM *** Set display & backdrop ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP 0
BACKDROP ON
REM *** Make the cube ***
MAKE OBJECT CUBE 1, 10
REM *** Cube to (50,0,0) after key
press ***
WAIT KEY
POSITION OBJECT 1,9,0,0
REM *** Move the cube backwards ***
WAIT KEY
POSITION OBJECT 1, 9,0,30
REM *** End program ***
WAIT KEY
END
```

Activity 31.5

```
REM *** Set display & backdrop ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP 0
BACKDROP ON
REM *** Make the set of objects ***
MAKE OBJECT CUBE 1,4
MAKE OBJECT BOX 2,10,15,5
MAKE OBJECT SPHERE 3,7
MAKE OBJECT CYLINDER 4,12
MAKE OBJECT CONE 5,12
REM *** Cube to (9,0,0) after key
press ***
WAIT KEY
POSITION OBJECT 1,9,0,0
REM *** Box to (-60,0,100) ***
WAIT KEY
POSITION OBJECT 2,-60,0,100
REM *** Sphere to (-30,-40,50) ***
WAIT KEY
POSITION OBJECT 3,-30,-40,50
REM *** Cylinder to (25,0,120) ***
WAIT KEY
POSITION OBJECT 4,25,0,120
REM *** Cone to (0,25,100) ***
WAIT KEY
POSITION OBJECT 5,0,25,100
REM *** End program ***
WAIT KEY
END
```

Activity 31.6

```
REM *** Set display & backdrop ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP 0
BACKDROP ON
REM *** Make the set of objects ***
MAKE OBJECT CUBE 1,10
REM *** Cube to(9,0,100) after key
press ***
WAIT KEY
POSITION OBJECT 1,9,0,100
REM *** Cube 31.3 units to the right
***
WAIT KEY
MOVE OBJECT RIGHT 1, 31.3
REM *** End program ***
WAIT KEY
END
```

Activity 31.7

```
Version 1
REM *** Set display & backdrop
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP 0
BACKDROP ON
REM *** Make the set of objects ***
MAKE OBJECT CUBE 1,40
REM ***Cube to(0,0,100) after key
press ***
POSITION OBJECT 1,0,0,100
REM *** Rotate cube ***
FOR degree = 1 TO 360
    XROTATE OBJECT 1, degree
    WAIT 1
NEXT degree
REM *** End program ***
WAIT KEY
END
```

```

Version 2
REM *** Set display & backdrop ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP 0
BACKDROP ON
REM *** Make the set of objects ***
MAKE OBJECT CUBE 1,40
REM ***Cube to(0,0,100) after key press ***
POSITION OBJECT 1,0,0,100
REM *** Rotate cube ***
FOR degree = 359 TO 0 STEP -1
  XROTATE OBJECT 1, degree
  WAIT 1
NEXT degree
REM *** End program ***
WAIT KEY
END

```

Activity 31.8

No solution required.

Activity 31.9

```

REM ** Set display mode ***
SET DISPLAY MODE 1280,1024,32
REM *** Create and position cube ***
MAKE OBJECT CUBE 1, 40
POSITION OBJECT 1, 0, 0, 200
REM *** Rotate cube 1 degree at a time ***
REM *** around all three axes ***
FOR angle = 1 TO 360
  ROTATE OBJECT 1, angle,angle,angle
  WAIT 10
NEXT angle
REM *** End program ***
WAIT KEY
END

```

Activity 31.10

```

REM *** Set screen resolution ***
SET DISPLAY MODE 1280,1024,32
REM *** Create and position cube ***
MAKE OBJECT CUBE 1, 40
POSITION OBJECT 1, 0, 0, 200
REM *** Revolve the cube ***
FOR c = 1 TO 360
  PITCH OBJECT UP 1, -1.0
  WAIT 10
NEXT c
REM *** End program ***
WAIT KEY
END

```

Activity 31.11

```

REM *** Set screen resolution ***
SET DISPLAY MODE 1280,1024,32
REM *** Create and position cube ***
MAKE OBJECT CUBE 1, 40
POSITION OBJECT 1, 0, 0, 200
REM *** Revolve the cube ***
FOR c = 1 TO 360
  TURN OBJECT RIGHT 1, 1.0
  WAIT 10
NEXT c
REM *** End program ***
WAIT KEY
END

```

Activity 31.12

```

REM *** Set screen resolution ***
SET DISPLAY MODE 1280,1024,32
REM *** Create and position cube ***
MAKE OBJECT CUBE 1, 40
POSITION OBJECT 1, 0, 0, 200
REM *** Revolve the cube ***
FOR c = 1 TO 360
  ROLL OBJECT LEFT 1, 1.0
  WAIT 10
NEXT c
REM *** End program ***
WAIT KEY
END

```

Activity 31.13

```

REM *** Set display & backdrop ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP 0
BACKDROP ON
REM *** Make the set of objects ***
MAKE OBJECT CUBE 1,40
REM ***Cube to(0,0,100) after key press ***
WAIT KEY
POSITION OBJECT 1,0,0,100
REM *** point cube at (-20,17,-10)***
WAIT KEY
POINT OBJECT 1,-20,17,-10
REM *** End program ***
WAIT KEY
END

```

Activity 31.14

The changes should cause the cube should move in the direction it is pointing.

Activity 31.15

No solution required.

Activity 31.16

The cube rotates in the same direction on both occasions.

Activity 31.17

```

REM *** Set display resolution ***
SET DISPLAY MODE 1280,1024,32
REM *** Make and position the cone ***
MAKE OBJECT CONE 1, 20
POSITION OBJECT 1, 0,0,200
REM *** Resize cone ***
SCALE OBJECT 1, 200.0,100.0,50.0
REM *** Rotate shape ***
DO
  TURN OBJECT RIGHT 1, 1.0
LOOP
REM *** End program ***
END

```

Activity 31.18

No solution required.

Activity 31.19

The second cube is created with the same rotation. The copied cube does not disappear when the original is deleted.

Activity 31.20

The duplicated cube is not rotated, but it is removed when the original cube is deleted.

When the second cube is deleted, the first cube is unaffected.

```
POSITION OBJECT 2, -25,0,100
DO
    x3D = MOUSEX() - SCREEN WIDTH() / 2
    y3D = -(MOUSEY() - SCREEN HEIGHT() / 2)
    POINT OBJECT 1, x3D,y3D,0
    POINT OBJECT 2, x3D,y3D,0
LOOP
REM *** End program ***
END
```

Activity 31.21

```
REM *** Set display resolution ***
SET DISPLAY MODE 1280,1024,32
REM *** Seed random number generator ***
RANDOMIZE TIMER()
REM *** Make and position the box ***
MAKE OBJECT BOX 1, RND(45)+5,RND(45)+5,
↳RND(45)+5
POSITION OBJECT 1, 0,0,0
REM *** Get dimensions of box ***
width# = OBJECT SIZE X(1)
height# = OBJECT SIZE Y(1)
depth# = OBJECT SIZE Z(1)
REM *** Display dimensions ***
DO
    SET CURSOR 10,20
    PRINT "Width : ",width#, "   Height :",
↳height#, "   Depth : ",depth#
LOOP
REM *** End program ***
WAIT KEY
END
```

Activity 31.22

```
REM *** Set display resolution ***
SET DISPLAY MODE 1280,1024,32
REM *** Seed random number generator ***
RANDOMIZE TIMER()
REM *** Make and position the box ***
MAKE OBJECT BOX 1, RND(45)+5,RND(45)+5,
↳RND(45)+5
POSITION OBJECT 1, 0,0,0
REM *** Rotate box at random ***
ROTATE OBJECT 1,RND(359),RND(359),RND(359)
REM *** Get dimensions of box ***
width# = OBJECT SIZE X(1)
height# = OBJECT SIZE Y(1)
depth# = OBJECT SIZE Z(1)
REM *** Get rotations of box ***
x_axis_rotation = OBJECT ANGLE X(1)
y_axis_rotation = OBJECT ANGLE Y(1)
z_axis_rotation = OBJECT ANGLE Z(1)
REM *** Display details ***
DO
    SET CURSOR 10,20
    PRINT "Width : ",width#, "   Height : "
↳height#, "   Depth : ",depth#
    SET CURSOR 10,40
    PRINT "X-axis : ",x_axis_rotation,
↳"   y-axis : ",y_axis_rotation,
↳"   z-axis : ",z_axis_rotation
LOOP
REM *** End program ***
WAIT KEY
END
```

Activity 31.23

```
REM *** Set display resolution ***
SET DISPLAY MODE 1280,1024,32
REM *** Make and position spheres ***
MAKE OBJECT SPHERE 1, 40
POSITION OBJECT 1,25,0,100
MAKE OBJECT SPHERE 2, 40
```

Activity 31.24

No solution required.

Activity 31.25

```
REM *** Set screen mode ***
SET DISPLAY MODE 1280,1024,32
REM *** Make and position cube ***
MAKE OBJECT CUBE 1, 40
POSITION OBJECT 1,25,0,100
REM *** Start in solid and culling on ***
wire = 0
cull = 1
REM *** Rotate cube ***
DO
    REM *** IF key pressed, switch mode ***
    IF INKEY$() = "w"
        wire = 1 - wire
        SET OBJECT WIREFRAME 1, wire
    ENDIF
    IF INKEY$() = "c"
        cull = 1 - cull
        SET OBJECT CULL 1,cull
    ENDIF
    PITCH OBJECT DOWN 1, 1.0
    TURN OBJECT LEFT 1, 1.0
LOOP
REM *** End program ***
END
```

Activity 31.26

The inside surfaces of both shapes are not drawn.

Activity 31.27

```
REM *** Set screen mode ***
SET DISPLAY MODE 1280,1024,32

REM *** Make & position cone and cylinder ***
MAKE OBJECT CONE 1, 5
POSITION OBJECT 1,-6,-5,0
XROTATE OBJECT 1, 45
MAKE OBJECT CYLINDER 2,5
POSITION OBJECT 2, 6,-5,0

REM ** Culling off for both objects ***
SET OBJECT CULL 1,0
SET OBJECT CULL 2,0

REM *** End program ***
WAIT KEY
END
```

Activity 31.28

In fact, the UNION operation only works predictably with cubes and boxes.

Activity 31.29

```
REM *** Set screen resolution and background ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP 0
BACKDROP ON
REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 0,0,-100
REM *** Create two shapes used ***
MAKE OBJECT CUBE 1,40
MAKE OBJECT BOX 2,10,30,10
POSITION OBJECT 2,0,15,0
REM *** Let viewer see position of shapes ***
WAIT KEY
REM *** Difference shapes ***
PERFORM CSG DIFFERENCE 1,2
REM *** Remove object 2 ***
DELETE OBJECT 2
REM *** Rotate new shape ***
DO
    TURN OBJECT LEFT 1,1.0
    PITCH OBJECT UP 1,1.0
LOOP
REM *** End program ***
END
```

Activity 31.30

```
REM *** Set screen resolution and background ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP 0
BACKDROP ON
REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 0,0,-100
REM *** Create two shapes used ***
MAKE OBJECT CUBE 1,40
MAKE OBJECT BOX 2,10,30,10
POSITION OBJECT 2,0,15,0
REM *** Let viewer see position of shapes ***
WAIT KEY
REM *** Intersection shapes ***
PERFORM CSG INTERSETION 1,2
REM *** Remove object 2 ***
DELETE OBJECT 2
REM *** Rotate new shape ***
DO
    TURN OBJECT LEFT 1,1.0
    PITCH OBJECT UP 1,1.0
LOOP
REM *** End program ***
END
```


Applying a Texture Image to a 3D Object

Colouring a 3D Object

Loading a Texture Image

Mipmaps

Offsetting a Texture

Overlaying Textures

Seamless Tiling

Semi-Transparent 3D Object

Sky Spheres

Texture Mapping Options

Texture Transparency

Tiling

Video Texturing

Adding Texture

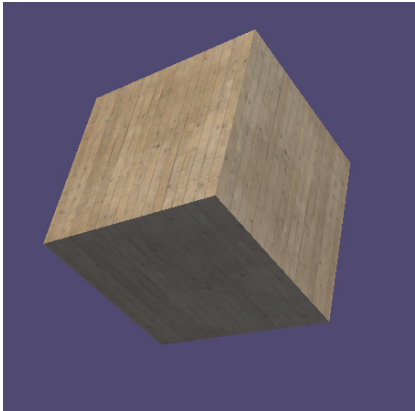
Introduction

The 3D shapes we have created so far look rather bland in white and shades of grey. To make things more interesting we can wrap an image around a 3D shape and thereby enhance its visual impact.

This is known as adding **texture** to the 3D object. An example of a textured cube is shown in FIG-32.1.

FIG-32.1

A Textured Cube



Loading a Texture Image

We need to start by loading the picture we intend to use to texture the 3D object into an image object with a statement such as:

```
LOAD IMAGE "texture01.bmp", 1
```

After this has been done, we can transfer the image to the surface of one or more 3D objects.

Using the Image as a Texture

The TEXTURE OBJECT Statement

A loaded image can become the texture of a 3D object by executing the TEXTURE OBJECT statement which has the format shown in FIG-32.2.

FIG-32.2

The TEXTURE
OBJECT Statement



In the diagram:

objno is an integer value specifying the object to which the texture is to be applied.

imgno is an integer value specifying the image to be used as the texture.

For example, we could apply image 1 to object 2 using the line

```
TEXTURE OBJECT 2,1
```

The program in LISTING-32.1 demonstrates how a cube can be textured with a wood image to create the impression of a wooden crate.

LISTING-32.1

Adding Texture to an
Object

```
REM *** Set display resolution ***
SET DISPLAY MODE 1280,1024,32

REM *** Load texture image ***
LOAD IMAGE "textureWood.jpg",1

REM *** Create cube ***
MAKE OBJECT CUBE 1,40

REM *** Add texture to cube ***
TEXTURE OBJECT 1,1

REM *** Position cube ***
POSITION OBJECT 1,25,0,100

REM *** Rotate cube continuously ***
DO
  TURN OBJECT LEFT 1, 1.0
LOOP

REM *** End program ***
END
```

Activity 32.1

Type in and test the program in LISTING-32.1 (*texture01.dbpro*).

Change the texture image to *eyecol.bmp*.

We can see quite clearly from the results of the last Activity that the image is applied separately to each face of the cube. For other shapes, the image may be applied differently.

Activity 32.2

Modify your last program so that *eyecol.bmp* is applied as a texture to a box, cylinder, cone and sphere (any dimensions will do). Create a separate program for each shape.

How often is the image repeated on each of the shapes?

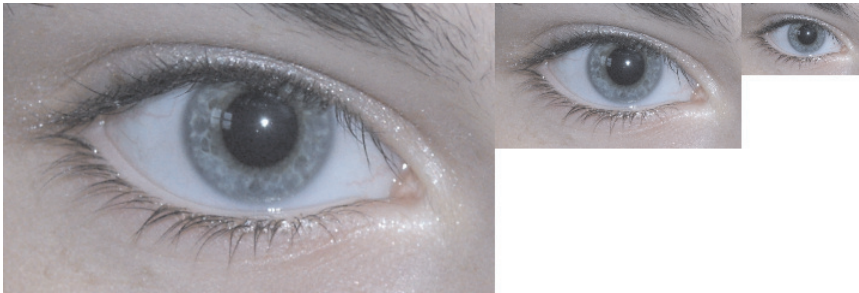
Mipmaps

Texturing a 3D object can be quite time consuming. It may be easy enough to map a 300 by 300 pixel image onto a flat surface which occupies exactly 300 by 300 pixels on the screen, but if the 3D object moves off into the distance, the computer has to work much harder to map the same 300 by 300 image onto an object which now occupies just 23 by 23 pixels on the screen.

To help with this problem DarkBASIC Pro creates more than one copy of any image that is loaded, with each copy being exactly half the size of the last (see FIG-32.3).

FIG-32.3

An Image with Added
Mipmaps



As a textured 3D object becomes smaller on the screen, the version of the image used to texture that object changes from the the largest to the smallest.

The program in LISTING-32.2 demonstrates this effect.

LISTING-32.2

Mipmaps in Action

```
REM *** Set display resolution ***
SET DISPLAY MODE 1280,1024,32

REM *** Load texture image ***
LOAD IMAGE "eyecol.bmp",1

REM *** Make cube ***
MAKE OBJECT CUBE 1,40
TEXTURE OBJECT 1,1
POSITION OBJECT 1,25,0,100

REM *** Move cube away from viewer ***
DO
    POSITION OBJECT 1,25,0,OBJECT POSITION Z(1)+10
LOOP

REM *** End program ***
END
```

Activity 32.3

Type in and test the program in LISTING-32.2 (*texture02.dbpro*).

The effect is a fairly subtle one. Look closely at the image on the cube as it moves away from your viewpoint. You should see it become less distinct as it gets smaller.

The LOAD IMAGE Statement Again

We met the LOAD IMAGE statement back in Chapter 20 when we created image objects which were then loaded into sprite objects. But the LOAD IMAGE statement has an expanded form which allows us to dictate whether mipmaps are to be created or not. This version of LOAD IMAGE has the format shown in FIG-32.4.

FIG-32.4

The LOAD IMAGE
Statement



In the diagram:

filename

is a string specifying the name of the file to be loaded.

imgno

is an integer specifying the ID to be allocated to the image object being created.

tflag

is an integer value specifying how the image is to be stored.

- 0 - mipmaps are created
- 1 - no mipmaps are created
- 2 - loads the image in as a cubemap texture (see the chapter on shaders)

When an image is loaded without mipmaps, any object using that image as a texture must continue to use the original image even when the 3D object is greatly reduced in size on the screen.

Activity 32.4

Modify your last program so that no mipmaps are used. To do this change the LOAD IMAGE line to read:

```
LOAD IMAGE "eyecol.bmp",1,1
```

How does the texture on the cube differ in this program from the earlier version?

Tiling

In the next example we'll create the floor of a dungeon by texturing a plane using a cobblestone image.

The program uses the following logic:

```
Load cobblestone image
Create large plane
Rotate plane to be horizontal
Texture plane using the cobblestone image
```

The program itself is given in LISTING-32.3.

LISTING-32.3

Creating a Floor Texture

```
REM *** Set display resolution ***
SET DISPLAY MODE 1280,1024,32
REM *** Load texture image ***
LOAD IMAGE "stonetile3.jpg",1
REM *** Create large plane ***
MAKE OBJECT PLAIN 1,400,400
REM *** Rotate plane to horizontal ***
XROTATE OBJECT 1,90
REM *** Texture plane ***
TEXTURE OBJECT 1,1

REM *** End program ***
WAIT KEY
END
```

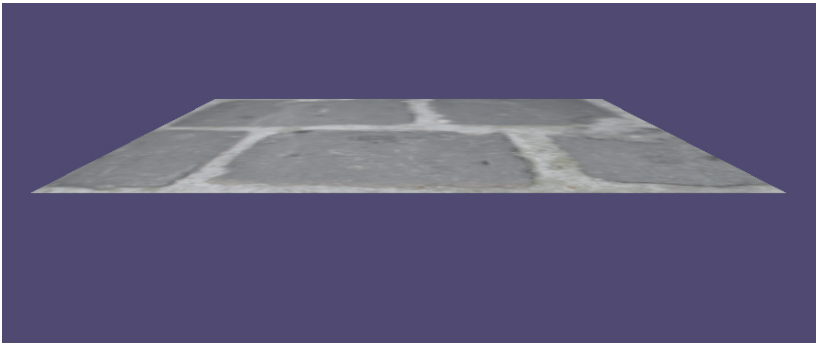
Activity 32.5

Type in and test the program in LISTING-32.3 (*texture03.dbpro*).

The screen dump in FIG-32.5 highlights the problem with the floor - the image has stretched over the whole plane giving a floor that contains only a few unrealistically large blocks rather than hundreds of smaller ones.

FIG-32.5

Floor Texturing



One way to solve the problem would be to use an image which actually shows the hundreds of blocks that we need to create a realistic floor. However, this may not be possible and the image would certainly have to be large if the visuals are to look convincing as a character moves over the floor.

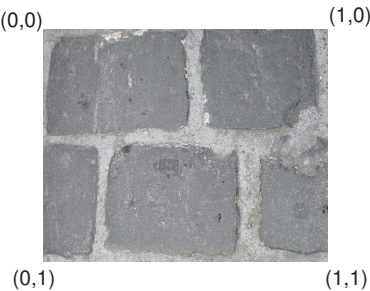
A second option is to make the texture image repeat itself several times over the surface of the plane. This, for rather obvious reasons, is known as **tiling**.

The SCALE OBJECT TEXTURE Statement

Any image employed as a texture uses a UV coordinate system with the top left being point (0,0) and the bottom right (1,1) no matter what the actual size of the image is (see FIG-32.6).

FIG-32.6

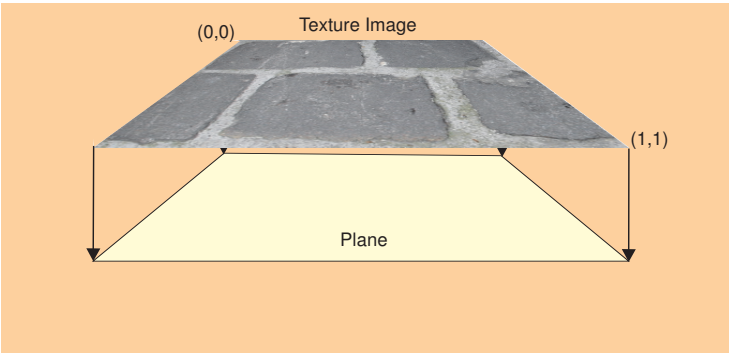
The UV Coordinate System used by a Texture Image



When mapped to a flat plane, the image spreads itself over the object with point (0,0) of the image mapping to the top-left corner of the plane and point (1,1) to the bottom-right corner (see FIG-32.7).

FIG-32.7

The Default Mapping of an Image to a Plane



Using the SCALE OBJECT TEXTURE statement, we can adjust this mapping making only a part of the image stretch over the whole object, or have the image duplicate itself several times creating a tiled effect as shown in FIG-32.8.

FIG-32.8

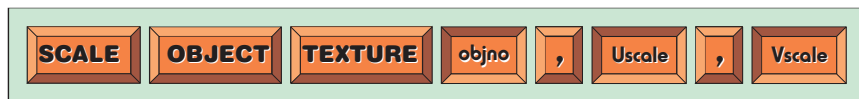
The Effects of Using
SCALE OBJECT
TEXTURE



The SCALE OBJECT TEXTURE statement has the format shown in FIG-32.9.

FIG-32.9

The SCALE OBJECT
TEXTURE Statement



In the diagram:

objno

is an integer value specifying the object to which the texture scaling is to be applied.

Uscale

is a real number specifying the multiplication factor along the U axis. Values less than 1 will result in only part of the image being used. Values greater than 1 will result in duplication of the image over the 3D object.

Vscale

is a real number specifying the multiplication factor along the V axis. Values less than 1 will result in only part of the image being used. Values greater than 1 will result in duplication of the image over the 3D object.

The first example shown in FIG-32.8 was created using the line:

```
SCALE OBJECT TEXTURE 1, 0.5, 0.5
```

while the second example was produced using:

```
SCALE OBJECT TEXTURE 1, 5.0, 5.0
```

The program in LISTING-32.4 demonstrates the effect of texture scaling by applying cobblestones texture repeated 10 times in each direction to the plane.

LISTING-32.4

Changing the Texture's
Scaling

```
REM *** Set display resolution ***
SET DISPLAY MODE 1280,1024,32
REM *** Load texture image ***
LOAD IMAGE "stonetile3.jpg",1
REM *** Create and position plain ***
MAKE OBJECT PLAIN 1, 400,400
XROTATE OBJECT 1, -90
REM *** Texture plain ***
TEXTURE OBJECT 1,1
REM *** Scale texture ***
SCALE OBJECT TEXTURE 1,10.0,10.0
REM *** End program ***
WAIT KEY
END
```

Activity 32.6

Type in and test the program given in LISTING-32.4 (*texture04.dbpro*).

Modify the scaling factors to each of the following settings and observe the results:

Uscale	Vscale
5.0	5.0
2.0	2.0
0.5	0.5
5.0	1.0
1.0	5.0

Change the 3D object used in your program from a plane to a sphere and retry each of the settings given above.

Scaling a texture image in this way affects the image itself, so there is no way to return to the original image settings within a program.

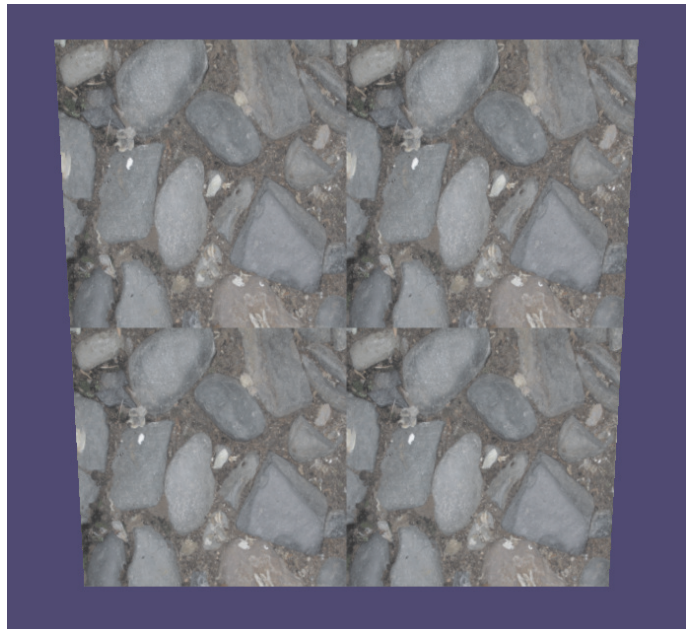
Seamless Tiling

For tiling to be convincing, the ends of the repeating image must butt together without too obvious a join.

If we start with a simple picture and use it as a tiled texture (as shown in FIG-32.10) we get a disappointing effect in which the edge of each image tile is very obvious.

FIG-32.10

A Visible Join Between
Tiles



To avoid this, we need to modify the image using a paint package such as Paint Shop Pro or Photoshop.

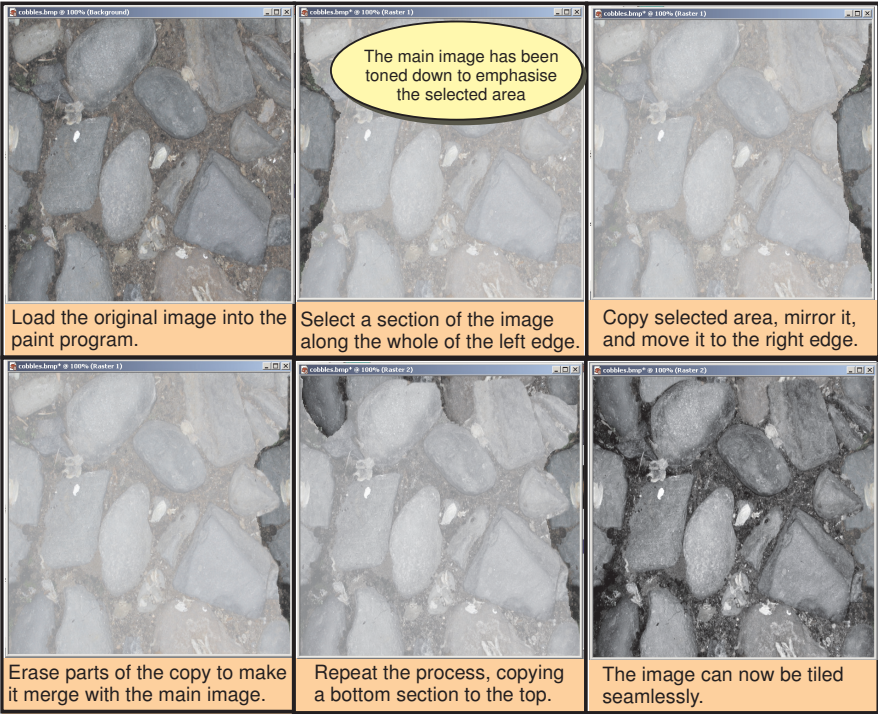
The stages involved are shown in FIG-32.11.

FIG-32.11

Creating an Image for
Seamless Tiling

Of course, it is equally possible to copy the right edge area to the left side and the top to the bottom.

You should choose whatever combinations suit the image in question.



It takes a bit of practice to achieve good results when creating a texture image, but the results can be worth it.

Even when an image is not tiled, we can still have a problem with seams. For example, when the image *eyecol.bmp* is applied as a texture to a sphere, the join between the left and right edges of the image is quite apparent at the back of the sphere, while the top and bottom edges are squeezed into single points at the two "poles".

Activity 32.7

Attempt to modify *eyecol.bmp* (creating a new file named *seamlesseye.bmp*) to give a seamless effect when textured onto a sphere.

Write a short program (*act3207.dbpro*) which applies the new file to a sphere and then rotates the sphere continuously about its local y-axis.

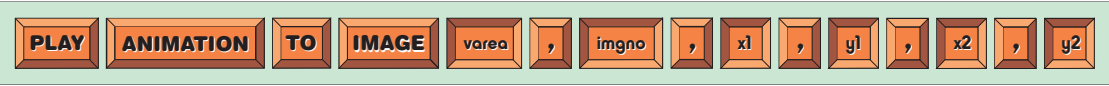
Video Texture

It is even possible to use a video clip as a surface texture. To do this we need to start by loading up a video with an instruction such as:

```
LOAD ANIMATION "mv1.mpg", 1
```

The PLAY ANIMATION TO IMAGE Statement

Now we need to transfer the video to an image object and this is done using the PLAY ANIMATION TO IMAGE statement which has the format shown in FIG-32.12.



In the diagram:

varea is an integer value specifying the video that is to be copied to an image area.

imgno is an integer specifying the image area to which the video is to be copied.

x1,y1,x2,y2 are the coordinates for the top-left and bottom right space which the video is to occupy.

The size of the play area (as set by *x1*, *y1*, *x2*, *y2*) affects the quality of the image when it appears on the 3D object; use too small a set of values and the video will be heavily pixellated; use too large a set of values and displaying the video will put too great a load on the processor/video card and slow the whole thing down.

Once the video has been transferred to the image object, we can then use the image to texture a 3D object in the usual manner. LISTING-32.5 demonstrates the effect by placing a video on a rotating cube.

LISTING-32.5

Using a Video as a Texture

```
REM *** Set display resolution ***
SET DISPLAY MODE 1280,1024,32

REM *** Load video ***
LOAD ANIMATION "lion.mpg",1

REM *** Transfer video to image ***
PLAY ANIMATION TO IMAGE 1,1,0,0,200,200

REM *** Create cube and texture with video ***
MAKE OBJECT CUBE 1,40
TEXTURE OBJECT 1,1

REM *** Move cube away from viewer ***
POSITION OBJECT 1,0,0,100

REM *** Rotate cube ***
DO
    TURN OBJECT LEFT 1,1.0
    PITCH OBJECT DOWN 1,1.0
LOOP

REM *** End program ***
END
```

Activity 32.8

Type in and test the program given in LISTING-32.5 (*texture05.dbpro*).

Modify the program to use very low values for the bottom right corner of the video (i.e. 10,10) and very high values (i.e. 1000,1000). What affect do these changes have on the final result?

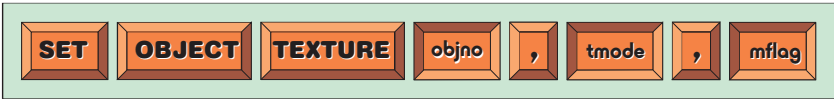
Other Texture Effects

The SET OBJECT TEXTURE Statement

An alternative way to achieve a seamless tiled texture is to use the SET OBJECT TEXTURE statement that adjusts the way in which each copy of the basic image is tiled onto the surface of a 3D object. The statement has the format shown in FIG-32.13.

FIG-32.13

The SET OBJECT TEXTURE Statement



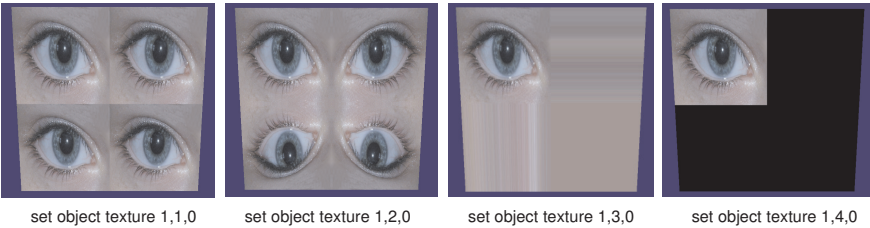
In the diagram:

<i>objno</i>	is an integer value specifying the object whose texture is to be modified.
<i>tmode</i>	is an integer value (1,2,3,4) which directly affects how the tiled texture is applied to the object. <ul style="list-style-type: none">1 - normal tiling.2 - images are mirrored/flipped so that identical edges meet.3 - The last pixel along each edge is extended over the remainder of the surface.4 - The image appears only once. The remainder of the surface is black.
<i>mflag</i>	Determines if mipmapping is to be used. <ul style="list-style-type: none">0 - mipmapping used1 - no mipmapping

In FIG-32.14 we see the effects of each possible value for *tmode* when applying *eyecol.bmp* as a tiled (2 by 2) texture on a cube.

FIG-32.14

The Effects of Using Different *tmode* Settings



The program in LISTING-32.6 shows a tile textured cube with the *tmode* value of the SET OBJECT TEXTURE statement set to 2.

LISTING-32.6

Using the SET OBJECT TEXTURE Statement

```
REM *** Set display resolution ***
SET DISPLAY MODE 1280,1024,32

REM *** Load image ***
LOAD IMAGE "eyecol.bmp",1

REM *** Create a plain ***
MAKE OBJECT PLAIN 1,200,200
```

continued on next page

LISTING-32.6
(continued)

Using the SET OBJECT
TEXTURE Statement

```
REM *** Texture object ***  
TEXTURE OBJECT 1,1  
SCALE OBJECT TEXTURE 1,2,2  
  
REM *** Modify tile mapping ***  
SET OBJECT TEXTURE 1,4,0  
  
REM *** End program ***  
WAIT KEY  
END
```

Activity 32.9

Type in and test the program in LISTING-32.6 (*texture06.dbpro*).
Try other settings for *tmode* and check the effects produced.

The SCROLL OBJECT TEXTURE Statement

The texture image can be mapped onto an object with a varying degree of offset along either the U or V axes. The overall effect is to modify which part of the texture image is placed at the top left corner of the object.

The effect is created using the SCROLL OBJECT TEXTURE statement which has the format shown in FIG-32.15.

FIG-32.15

The SCROLL OBJECT
TEXTURE Statement



In the diagram:

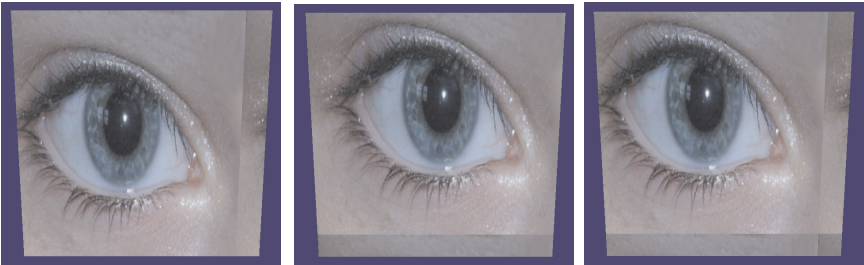
objno is an integer value specifying the object whose texture is to be scrolled.

Uoffset, Voffset are a pair of real values representing the coordinates of the image that are to be the top-left corner of the texture when placed on a 3D object.

Examples of this statement in use are shown in FIG-32.16 where *eyecol.bmp* is mapped to a plane object with various *Uoffset, Voffset* values.

FIG-32.16

The Effects of the
SCROLL OBJECT
TEXTURE Statement



scroll object texture 1,0.1,0.0 scroll object texture 1,0.0,0.1 scroll object texture 1,0.1,0.1

A complete program demonstrating the effect is given in LISTING-32.7.

LISTING 32.7

Using the SCROLL
OBJECT TEXTURE
Statement

```
REM *** Set display resolution ***
SET DISPLAY MODE 1280,1024,32

REM *** Load image ***
LOAD IMAGE "eyecol.bmp",1

REM *** Create and texture plane object ***
MAKE OBJECT PLAIN 1,200,200
TEXTURE OBJECT 1,1

REM *** Offset the texture placed on the image***
SCROLL OBJECT TEXTURE 1,0.1,0.0

REM *** End program ***
WAIT KEY
END
```

Activity 32.10

Type in and test the program in LISTING-32.7 (*texture07.dbpro*).

The effect makes a permanent change to the texture for that image, so repeating the same statement creates a further offset.

Activity 32.11

In your last program, add another SCROLL OBJECT TEXTURE statement immediately after the first using the same values.

How does this affect the texture?

By placing the SCROLL OBJECT TEXTURE statement in a loop, the texture can scroll over the surface of the 3D object.

Activity 32.12

Remove the second SCROLL OBJECT TEXTURE statement from your last program and insert the remaining SCROLL OBJECT TEXTURE statement in a DO..LOOP structure.

What affect does this have on the 3D object's texture?

Modify the SCROLL OBJECT TEXTURE statement so that the texture scrolls horizontally rather than vertically.

Activity 32.13

In Chapter 31 you created a program in which two spheres followed the movement of the mouse pointer (*object3D11.dbpro*).

Modify that program so that the spheres are textured using *seamlesseye.bmp*.

Add the appropriate SCROLL OBJECT TEXTURE statements so that the pupils of the eyes face the mouse pointer.

The SET OBJECT TRANSPARENCY Statement

A colour other than black can become the transparent colour using the SET COLOR KEY statement.

When an image containing black is mapped to a sprite, any black areas in the image are automatically transparent when the sprite appears on the screen. However, this is not the case with 3D objects.

To demonstrate this, the next program (see LISTING-32.8) uses the image shown in FIG-32.17 as the texture on a rotating cube.

FIG-32.17

Image used to Texture a Cube



LISTING-32.8

Black Textured Areas are not Transparent on a 3D Object

```
REM *** Set display resolution ***
SET DISPLAY MODE 1280,1024,32

REM *** Load texture image ***
LOAD IMAGE "DoNot.bmp",2

REM *** Make and position cube ***
MAKE OBJECT CUBE 1, 40
POSITION OBJECT 1,25,0,100

REM *** Texture cube with image ***
TEXTURE OBJECT 1,2

REM *** Rotate cube ***
DO
    PITCH OBJECT DOWN 1, 1.0
    TURN OBJECT LEFT 1, 1.0
LOOP

REM *** End program ***
END
```

Activity 32.14

Type in and test the program given in LISTING-32.8 (*texture08.dbpro*).

However, we can force a 3D object to make black (or whatever other colour as been set as the background colour using the SET COLOR KEY statement) transparent using the SET OBJECT TRANSPARENCY statement which has the format shown in FIG-32.18.

FIG-32.18

The SET OBJECT TRANSPARENCY



In the diagram:

objno

is an integer value specifying the object whose background texture colour is to be made transparent.

transflag

is 0 or 1.

- 0 - background colour not transparent.
- 1 - background colour transparent.

Activity 32.15

Modify your last program by adding the line

```
SET OBJECT TRANSPARENCY 1,1
```

immediately before the DO..LOOP structure.

How does this affect the appearance of the cube.

The SET DETAIL MAPPING ON Statement

A second image can be combined with the basic texture image of an object to create a new texture consisting of both images.

For example, if we take the images shown in FIG-32.19 with image 1 being the basic texture and image 2 the overlaid texture, then we achieve the effect shown in FIG-32.20 when these are applied to a cube.

FIG-32.19

The Images Used To
Texture a 3D Object



Image 1

Image 2

FIG-32.20

The Two Images Applied
to a Cube

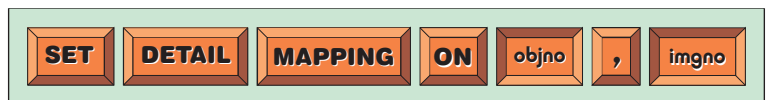


Notice that any black areas in image 2 are automatically transparent.

A second image is applied to the texture of an object using the SET DETAIL MAPPING ON statement which has the format shown in FIG-32.21.

FIG-32.21

The SET DETAIL
MAPPING ON Statement



In the diagram:

objno

is an integer value specifying the object to which
a second texture image is to be added.

imgno

is an integer value specifying the image object containing the picture to be used as a second texture.

LISTING-32.9 creates the rotating cube shown above. The main lines of code are

```
LOAD IMAGE "DoNot.bmp", 2
```

which loads the secondary image being used and

```
SET DETAIL MAPPING ON 1, 2
```

which applies this image as an overlaid texture.

LISTING-32.9

Using a Secondary
Texture on a 3D Object

```
REM *** Set display resolution ***
SET DISPLAY MODE 1280,1024,32
REM *** Load texture images ***
LOAD IMAGE "textureWood.jpg", 1
LOAD IMAGE "DoNot.bmp", 2
REM *** Create and texture cube ***
MAKE OBJECT CUBE 1, 40
TEXTURE OBJECT 1,1

REM *** Add secondary texture ***
SET DETAIL MAPPING ON 1,2

REM *** Position cube ***
POSITION OBJECT 1,25,0,100

REM *** Rotate cube ***
DO
    PITCH OBJECT DOWN 1, 1.0
    TURN OBJECT LEFT 1, 1.0
LOOP
REM *** End program ***
END
```

Activity 32.16

Type in and test the program given in LISTING-32.9 (*texture09.dbpro*).

Replace *DoNot.bmp* with *DoNotMag.bmp* which contains black text on a magenta background. Set magenta as the transparent colour using the SET IMAGE COLORKEY statement.

We are limited to a single image when overlaying an object's texture with detail. So, attempting to add a second detail image will simply remove the first from the object.

Activity 32.17

Add *FlagMag.bmp* as a second detail image to the cube object in your last program.

What effect does this create?

If an object's texture has been tiled using the SCALE OBJECT TEXTURE statement, any additional image added using SET DETAIL MAPPING ON will also be tiled to the same extent as the original texture.

Activity 32.18

In your last program, remove all references to the *flagmag.bmp* file. Create a tiled effect on the cube by using the **SCALE OBJECT TEXTURE** statement with the *Uscale* and *Vscale* parameters both set to 2.

How is the **DETAIL MAPPING** image on the cube affected by the tiling?

The SET OBJECT FILTER Statement

Different methods of texturing can be specified using the **SET OBJECT FILTER** statement. The differences achieved have little obvious effect on the visible appearance of the textured object itself, but modify the algorithm used to create that texture. The statement has the format shown in FIG-31-22.

FIG-32.22

The SET OBJECT
FILTER Statement



In the diagram:

objno

is an integer value specifying the object whose texture is to be filtered.

filterflag

is 0, 1, or 2

- 0 - always uses the original image to texture (it never uses the smaller images created by mipmapping).
- 1 - no smoothing is used.
- 2 - uses linear filtering.

The program in LISTING-32.10 creates 3 spheres, each textured using one of the filter options. You may see a slight difference in the appearance of the spheres as they move off into the background and reduce in size.

LISTING-32.10

Using the SET OBJECT
FILTER Statement

```
REM *** Set screen resolution ***
SET DISPLAY MODE 1280,1024,32
REM *** Load image used as texture ***
LOAD IMAGE "grid8by8.bmp",1,1
REM *** Create three spheres ***
MAKE OBJECT SPHERE 1 ,40
MAKE OBJECT SPHERE 2 ,40
MAKE OBJECT SPHERE 3 ,40

REM *** Texture each sphere ***
TEXTURE OBJECT 1,1
TEXTURE OBJECT 2,1
TEXTURE OBJECT 3,1

REM *** Set different filter for each sphere ***
SET OBJECT FILTER 1, 0
SET OBJECT FILTER 2, 1
SET OBJECT FILTER 3, 2

REM *** Position spheres ***
POSITION OBJECT 1,-42,0,0
POSITION OBJECT 2,0,0,0
POSITION OBJECT 3,42,0,0
```

continued on next page

LISTING-32.10

(continued)

Using the SET OBJECT
FILTER Statement

```
REM *** Moves spheres ***
FOR z = 1 TO 4000
    MOVE OBJECT 1,1
    MOVE OBJECT 2,1
    MOVE OBJECT 3,1
    WAIT 10
NEXT z

REM *** End program ***
WAIT KEY
END
```

Activity 32.19

Type in and test the program given above (*texture10.dbpro*).

Summary

- Texturing involves mapping an image onto the surface of a 3D object.
- Use LOAD IMAGE to load any image which is to be used to texture a 3D object.
- Use TEXTURE OBJECT to map an image to an object.
- Images stretch automatically to fit the surface of an object.
- On a cube or box the image is repeated on each face.
- On other 3D objects, the image appears only once.
- Mipmaps are smaller versions of the original image which are used to speed up mapping when a textured object becomes much smaller than the original image.
- Tiling is the application of an image multiple times to the same surface.
- Use SCALE OBJECT TEXTURE to create a tiled texture.
- To create a seamless tile, make sure the opposite edges are complementary.
- Use PLAY ANIMATION TO IMAGE and TEXTURE OBJECT to display a video on the surface of an object.
- Use SET OBJECT TEXTURE to specify how an image is mapped to a surface.
- Use SCROLL OBJECT TEXTURE to create an offset mapping of the image on a 3D surface.
- Use SET OBJECT TRANSPARENCY to make black areas of a 3D object disappear.
- Use SET DETAIL MAPPING ON to apply a second image to an already textured object.
- Use SET OBJECT FILTER to modify how an image is filtered when being mapped onto an object.

Other Visual Effects

Introduction

Although adding texture to a 3D object is the commonest way of changing an object's appearance, it is by no means the only option available. In this section we'll see some other options that are available to us in DarkBASIC Pro.

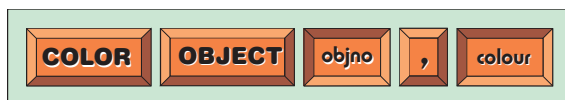
Changing Colour and Transparency

The COLOR OBJECT Statement

Rather than add a texture to a 3D object, we can give it a surface of a specific colour using the COLOR OBJECT statement which has the format shown in FIG-32.23.

FIG-32.23

The COLOR OBJECT Statement



In the diagram:

objno

is an integer value specifying the object which is to be coloured.

colour

is an integer value specifying the colour to be used on the object's surface.

For example, we could give object 1 a red surface using the line:

```
COLOR OBJECT 1, RGB(255,0,0)
```

An example of this statement in operation is given in LISTING-32.11 which colours a rotating cube in red, changing to green when a random event occurs.

LISTING-32.11

Using COLOR OBJECT

```
REM *** Set display resolution ***
SET DISPLAY MODE 1280,1024,32

REM *** Seed random number generator ***
RANDOMIZE TIMER()
REM *** Make and position cube ***
MAKE OBJECT CUBE 1, 40
POSITION OBJECT 1,25,0,100
REM *** Colour cube red ***
COLOR OBJECT 1, RGB(255,0,0)
REM *** Rotate Cube ***
DO
    PITCH OBJECT DOWN 1, 1.0
    TURN OBJECT LEFT 1, 1.0
    REM *** One chance in 1000 of changing to green ***
    IF RND(1000) = 500
        COLOR OBJECT 1, RGB(0,255,0)
    ENDIF
LOOP
REM *** End program ***
END
```

Activity 32.20

Type in and test the program given in LISTING-32.11 (*texture11.dbpro*).

A coloured surface cannot be used in conjunction with a main texture, but secondary textures (created using SET DETAIL MAPPING ON) may still be used.

Activity 32.21

Modify your last program so that *DoNot.bmp* is used as a secondary texture on the surface of the cube.

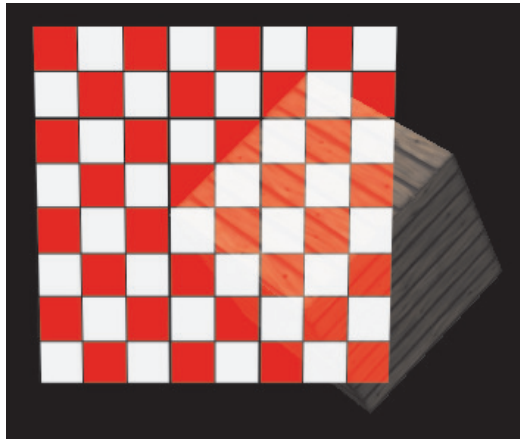
The GHOST OBJECT ON Statement

In FIG-32.24 we can see a cube which is semi-transparent with the grided plane in the background showing through the cube.

FIG-32.24

A Transparent Cube

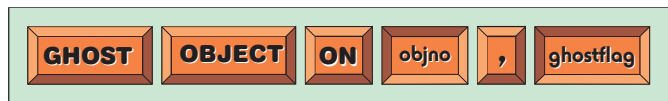
You'll have to look closely to see the cube!



We can create this effect using the GHOST OBJECT ON statement which has the format shown in FIG-32.25.

FIG-32.25

The GHOST OBJECT ON Statement



In the diagram:

objno

is an integer value specifying the 3D object to be made semi-transparent.

ghostflag

is 0 to 5.

- 0 - object is semi-transparent
- 1 - object uses negative transparency
- 2 - object is semi-transparent but lighter
- 3 - uses the image's alpha channel
- 4 - similar to 1 but lighter
- 5 - object is opaque

The program in LISTING-32.12 demonstrates the effect of this instruction by rotating the cube with a textured plane in the background.

LISTING-32.12

Using the GHOST
OBJECT ON Statement

```
REM *** Set display resolution ***
SET DISPLAY MODE 1280,1024,32

REM *** Load texture images ***
LOAD IMAGE "grid8by8.bmp",1
LOAD IMAGE "DoNot.bmp",2

REM *** Make and position cube ***
MAKE OBJECT CUBE 1, 40
POSITION OBJECT 1,25,0,100

REM *** Texture cube ***
TEXTURE OBJECT 1,1

REM *** Create background plane ***
MAKE OBJECT PLAIN 2,100,100
TEXTURE OBJECT 2, 2
POSITION OBJECT 2,0,0,200

REM *** Create semi-transparent cube ***
GHOST OBJECT ON 1,0

REM *** Rotate cube ***
DO
    PITCH OBJECT DOWN 1, 1.0
    TURN OBJECT LEFT 1, 1.0
LOOP

REM *** End program ***
END
```

Activity 32.22

Type in and test the program given in LISTING-32.12 (*texture12.dbpro*).
Modify the *transflag* value in the GHOST OBJECT ON statement and observe the effects of the various settings.

The GHOST OBJECT OFF Statement

The semi-transparency effect created by GHOST OBJECT ON can be disabled using the GHOST OBJECT OFF statement which has the format shown in FIG-32.26.

FIG-32.26

The GHOST OBJECT
OFF Statement



In the diagram:

objno

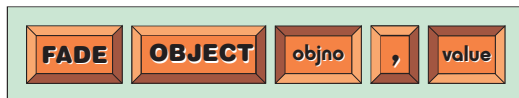
is an integer value specifying the 3D object in which the semi-transparency mode is switched off.

The FADE OBJECT Statement

The amount of light reflected from the surface of a 3D object can be modified using the FADE OBJECT statement. This allows settings varying between no light reflected and twice normal reflection. The statement has the format shown in FIG-32.27.

FIG-32.27

The FADE OBJECT
Statement



In the diagram:

objno

is an integer value specifying the 3D object whose reflective index is to be changed.

value

is an integer value between 0 and 200. A value of zero means that the object will reflect no light; a value of 100 creates normal amount of reflected light; a value of 200 gives twice the normal amount of reflected light.

The program in LISTING-32.13 demonstrates the effect of this statement by gradually reducing the reflective value from 200 to zero.

LISTING-32.13

Using FADE OBJECT to
make a Transparent
Object Disappear

```
REM *** Set display resolution ***
SET DISPLAY MODE 1280,1024,32

REM *** Load texture images ***
LOAD IMAGE "textureWood.jpg",1
LOAD IMAGE "DoNot.bmp",2

REM *** Make and position cube ***
MAKE OBJECT CUBE 1, 40
POSITION OBJECT 1,25,0,100

REM *** Texture cube ***
TEXTURE OBJECT 1,1

REM *** Create background plain ***
MAKE OBJECT PLAIN 2,100,100
TEXTURE OBJECT 2, 2
POSITION OBJECT 2,0,0,200

REM *** Start reflective value at 200 ***
reflectivity = 200

REM *** rotate cube ***
DO
    PITCH OBJECT DOWN 1, 1.0
    TURN OBJECT LEFT 1, 1.0
    FADE OBJECT 1,reflectivity
    REM *** Decrement reflective value until it reaches zero ***
    IF reflectivity > 0
        DEC reflectivity
    ENDIF
    WAIT 10
LOOP

REM *** End program ***
END
```

Activity 32.23

Type in and test the program given in LISTING-32.13 (*texture13.dbpro*).

When used in combination with the GHOST OBJECT ON statement, the FADE OBJECT statement can make a semi-transparent object disappear completely.

Activity 32.24

In your last program, add the line

```
GHOST OBJECT ON 1,0
```

immediately before the DO..LOOP structure.

How does this affect the cube?

There's still more to be said about texturing objects but we'll leave that to a later chapter after we've covered other basic concepts such as cameras and lighting.

Summary

- Use COLOR OBJECT to tint the surface of a 3D object.
- Use GHOST OBJECT ON to make an object transparent.
- Use GHOST OBJECT OFF to make a transparent object opaque.
- Use FADE OBJECT to reduce the amount of light reflected by an object.
- When used on a transparent object, FADE OBJECT can make that object invisible.
- When used on an opaque object, FADE OBJECT can make that object completely black.

Images with an Alpha Channel

Introduction

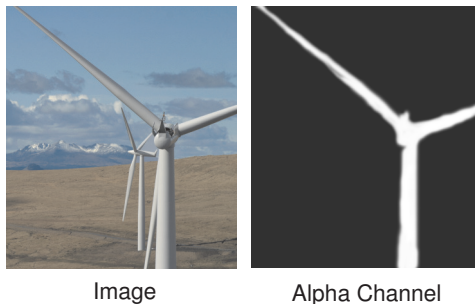
As we saw back in Chapter 30, some picture files contain an alpha channel which can affect the visibility of the main image. This type of picture file, when used as a texture, affects the overall result obtained.

Using Images with an Alpha Channel

The image *windmillshaped.tga* contains an image and an alpha channel as shown in FIG-32.28.

FIG-32.28

An Image and its Alpha Channel



We've already seen that the SET OBJECT TRANSPARENCY statement affects the black area of an texture image, but the statement also controls how an alpha channel within an image affects the final texture.

The program in LISTING 32-14 textures a cube using *windmillshaped.tga*. With the default settings, the alpha channel within the image has no effect on the texturing of the cube, but after a SET OBJECT TRANSPARENCY statement is used to modify the texturing, the alpha channel changes the final look of the cube.

LISTING-32.14

Texturing with
Alpha-Channel Images

```
REM *** Set display resolution ***
SET DISPLAY MODE 1280, 1024, 32

REM *** Create cube ***
MAKE OBJECT CUBE 1,10

REM *** Texture cube ***
LOAD IMAGE "windmillshaped.tga",1
TEXTURE OBJECT 1,1

REM *** Rotate cube ***
DO
  REM *** IF key pressed, use alpha channel ***
  IF INKEY$() <> ""
    SET OBJECT TRANSPARENCY 1,1
  ENDIF
  TURN OBJECT LEFT 1,1
LOOP
REM *** End program ***
END
```

Activity 32.25

Type in and test the program in LISTING-32.14 (*texture14.dbpro*).

Notice that the other parts of the image do not disappear completely when the alpha channel is activated. This is because the darkened areas of the alpha channel are grey and not black. If black had been used, all other parts of the image would have become invisible.

An image with an alpha channel also produces an effect when option 3 is used with the GHOST OBJECT ON statement.

Activity 32.26

In your last program, change the line

```
SET OBJECT TRANSPARENCY 1,1
```

to

```
GHOST OBJECT ON 1,3
```

Observe how this affect the program's display.

Summary

- Use SET OBJECT TRANSPARENCY with an alpha channel image to create transparent or semitransparent texturing effects.
- Use GHOST OBJECT ON with option 3 to make use of the alpha channel information in creating the final ghosting effect.

Creating a Complex 3D Shape

Introduction

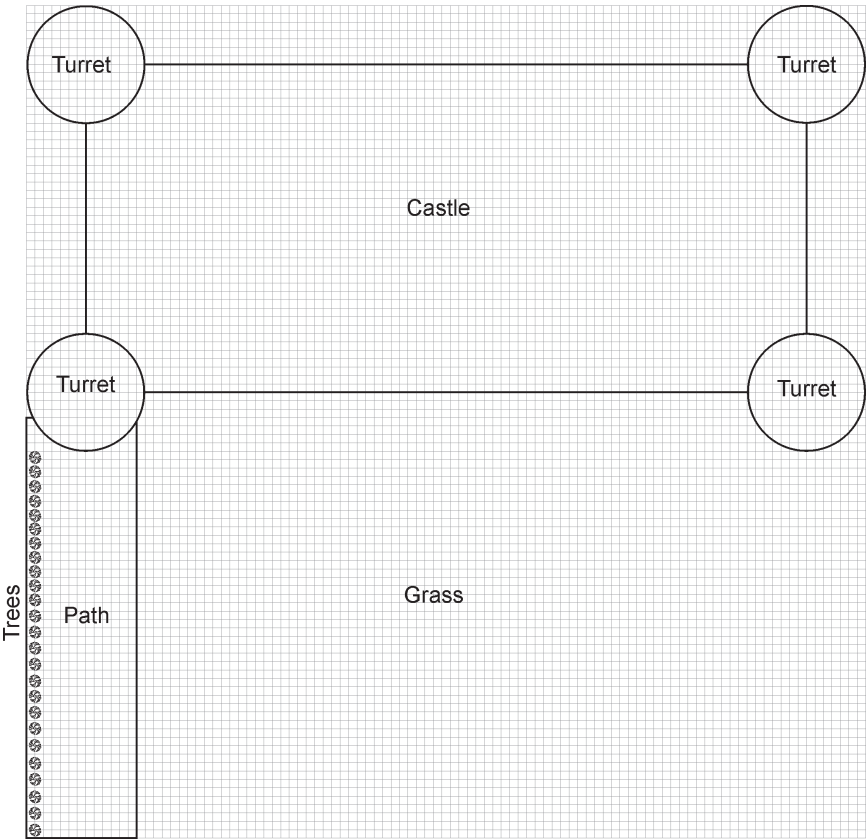
By combining the basic 3D shapes available to us we can create almost any complex shape. However, this may take a considerable amount of work and require a great deal of code. In reality we would probably create objects such as these in a separate 3D drawing package and then import the resulting file into our DarkBASIC Pro program (we'll see how to do this in Chapter 36). However, just to give us some practice, we'll try creating a simple castle using only DarkBASIC Pro.

Designing the Castle

We are going to save ourselves a great deal of time later if we start by doing a grided plan of the castle to give ourselves the basic layout and sizes. FIG-32.29 shows a plan of the castle.

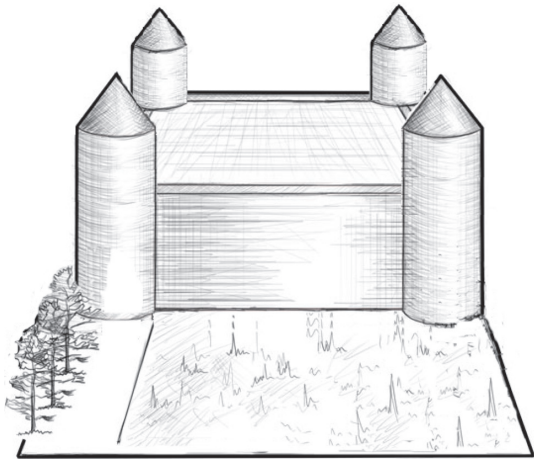
FIG-32.29

A Rough Sketch of the Object Required



Next, we need to create a more traditional drawing showing the characteristics of the castle (see FIG-32.30). We might also draw specific parts in more detail.

FIG-32.30
A More Detailed
Design of the Castle



Gathering the Components

The actual texture files being used need to be obtained or created. If you're not much of an artist, then you'll find plenty of texture files on the Internet, but if you're intending to create a commercial product, remember that almost everything you see on the Internet will be owned by someone and they expect to be paid if you are going to use their material. Even material that is advertised as free may still need to be paid for when used in a commercial product.

The textures used on the castle are shown in FIG-32.31.

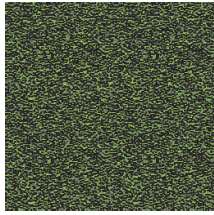
FIG-32.31
Textures Used



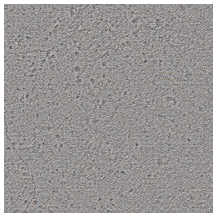
Filename : CobbleStones.jpg
Object : path



Filename : stone.jpg
Object : castle walls and turrets



Filename : grass1.jpg
Object : lawn



Filename : ceil_U3_01.jpg
Object : castle ceiling



Filename : rock2.jpg
Object : castle floor



Filename : tiles.jpg
Object : castle and turret roofs



Filename : tree.jpg
Object : trees lining path

Creating the Code

The coding (see LISTING-32.15) is long but fairly straight forward. The complete castle is created by a function, but this calls other functions which draw the various parts of the castle.

Splitting the code in this way will help us keep the structure as understandable as possible.

LISTING-32.15

Drawing the Castle

```

REM *** Building Components ***
REM *** Texture Images ***
#CONSTANT grass      1
#CONSTANT road       2
#CONSTANT tree        3
#CONSTANT flooring    4
#CONSTANT wall        5
#CONSTANT roofing     6
#CONSTANT cover       8
#CONSTANT trellis     9
#CONSTANT transport  10
REM *** 3D Objects ***
#CONSTANT lawn        1
#CONSTANT approach    2
#CONSTANT floor1      3
#CONSTANT ceiling     5
#CONSTANT tree1       7
#CONSTANT frontwall   51
#CONSTANT backwall    52
#CONSTANT leftwall    53
#CONSTANT rightwall   54
#CONSTANT innerwall   55
#CONSTANT turret1     56
#CONSTANT turretroof1 57
#CONSTANT turret2     58
#CONSTANT turretroof2 59
#CONSTANT turret3     60
#CONSTANT turretroof3 61
#CONSTANT turret4     62
#CONSTANT turretroof4 63
#CONSTANT roof1       64
#CONSTANT column      70

REM *** Set screen resolution ***
SET DISPLAY MODE 1280,1024,32
DrawCastle()
WAIT KEY
END

FUNCTION DrawCastle()
    LoadImages()
    DrawGrounds()
    DrawExternalWalls()
    DrawRoofandCeiling()
    DrawTurrets()
    DrawInternalColumns()
ENDFUNCTION

FUNCTION LoadImages()
    REM *** Load texture images ***
    LOAD IMAGE "grass1.jpg",lawn
    LOAD IMAGE "CobbleStones.jpg",road
    LOAD IMAGE "tree.jpg",tree
    LOAD IMAGE "rock2.jpg",flooring
    LOAD IMAGE "stone.jpg",wall
    LOAD IMAGE "tiles.jpg",roofing
    LOAD IMAGE "ceil_U3_01.jpg",cover
ENDFUNCTION

```

continued on next page

LISTING-32.15

(continued)

Drawing the Castle

```
FUNCTION DrawGrounds()  
    REM *** Create lawn ***  
    MAKE OBJECT PLAIN lawn, 500,700  
    TEXTURE OBJECT lawn, grass  
    SCALE OBJECT TEXTURE lawn, 100,100  
    XROTATE OBJECT lawn,-90  
    POSITION OBJECT lawn,250,0,350  
    SET DETAIL MAPPING ON lawn,transport  
    REM *** Create approach road ***  
    MAKE OBJECT PLAIN approach, 50,700  
    TEXTURE OBJECT approach,road  
    SCALE OBJECT TEXTURE approach,8,50  
    XROTATE OBJECT approach,-90  
    POSITION OBJECT approach, -25,0,350  
    REM *** Create castle floor ***  
    MAKE OBJECT PLAIN floor1, 550,300  
    TEXTURE OBJECT floor1, flooring  
    SCALE OBJECT TEXTURE floor1,100,100  
    XROTATE OBJECT floor1,-90  
    POSITION OBJECT floor1,225,0,850  
    DrawTrees()  
ENDFUNCTION  
  
FUNCTION DrawExternalWalls()  
    REM *** Create front wall ***  
    MAKE OBJECT PLAIN frontwall,550,100  
    TEXTURE OBJECT frontwall,wall  
    SCALE OBJECT TEXTURE frontwall,30,6  
    POSITION OBJECT frontwall, 225,50,700  
  
    REM *** Create back wall ***  
    MAKE OBJECT PLAIN backwall,550,100  
    TEXTURE OBJECT backwall,wall  
    SCALE OBJECT TEXTURE backwall,50,10  
    POSITION OBJECT backwall, 225,50,1000  
  
    REM *** Create left wall ***  
    MAKE OBJECT PLAIN leftwall,300,100  
    TEXTURE OBJECT leftwall,wall  
    SCALE OBJECT TEXTURE leftwall,30,10  
    YROTATE OBJECT leftwall, -90  
    POSITION OBJECT leftwall,-50,50,850  
  
    REM *** Create right wall ***  
    MAKE OBJECT PLAIN rightwall,300,100  
    TEXTURE OBJECT rightwall,wall  
    SCALE OBJECT TEXTURE rightwall,30,10  
    YROTATE OBJECT rightwall,90  
    POSITION OBJECT rightwall,500,50,850  
ENDFUNCTION  
  
FUNCTION DrawTurrets()  
    REM *** Create first turret ***  
    MAKE OBJECT CYLINDER turret1,200  
    SCALE OBJECT turret1,40,100,40  
    MAKE OBJECT CONE turretroof1,81  
    REM *** Texture turret ***  
    TEXTURE OBJECT turret1,wall  
    SCALE OBJECT TEXTURE turret1,10,10  
    TEXTURE OBJECT turretroof1,roofing  
    SCALE OBJECT TEXTURE turretroof1,5,10  
    REM *** Position turret ***  
    POSITION OBJECT turret1, -25,100,970  
    POSITION OBJECT turretroof1,-25,240,970
```

continued on next page

LISTING-32.15

(continued)

Drawing the Castle

```
REM *** Second turret ***
MAKE OBJECT CYLINDER turret2,200
SCALE OBJECT turret2,40,100,40
MAKE OBJECT CONE turretroof2,81
REM *** Texture turret ***
TEXTURE OBJECT turret2,wall
SCALE OBJECT TEXTURE turret2,10,10
TEXTURE OBJECT turretroof2,roofing
SCALE OBJECT TEXTURE turretroof2,5,10
REM *** Position turret ***
POSITION OBJECT turret2, 475,100,970
POSITION OBJECT turretroof2,475,240,970
REM *** Third turret ***
MAKE OBJECT CYLINDER turret3,200
SCALE OBJECT turret3,40,100,40
MAKE OBJECT CONE turretroof3,81
REM *** Texture turret ***
TEXTURE OBJECT turret3,wall
SCALE OBJECT TEXTURE turret3,10,10
TEXTURE OBJECT turretroof3,roofing
SCALE OBJECT TEXTURE turretroof3,5,10
REM *** Position turret ***
POSITION OBJECT turret3, -25,100,725
POSITION OBJECT turretroof3,-25,240,725
REM *** Fourth turret ***
MAKE OBJECT CYLINDER turret4,200
SCALE OBJECT turret4,40,100,40
MAKE OBJECT CONE turretroof4,81
REM *** Texture turret ***
TEXTURE OBJECT turret4,wall
SCALE OBJECT TEXTURE turret4,10,10
TEXTURE OBJECT turretroof4,roofing
SCALE OBJECT TEXTURE turretroof4,5,10
REM *** Position turret ***
POSITION OBJECT turret4, 475,100,725
POSITION OBJECT turretroof4,475,240,725
ENDFUNCTION

FUNCTION DrawRoofAndCeiling()
REM *** Create main roof ***
MAKE OBJECT PLAIN roof1,550,300
REM *** Texture main roof ***
TEXTURE OBJECT roof1,roofing
SCALE OBJECT TEXTURE roof1,50,10
REM *** Position roof ***
XROTATE OBJECT roof1, -90
POSITION OBJECT roof1,225,100,850
REM *** Create ceiling ***
MAKE OBJECT PLAIN ceiling,550,300
REM *** Texture ceiling ***
TEXTURE OBJECT ceiling,cover
SCALE OBJECT TEXTURE ceiling,5,2
REM *** Position ceiling ***
XROTATE OBJECT ceiling, -90
POSITION OBJECT ceiling,225,99,850
ENDFUNCTION

FUNCTION DrawInternalColumns()
RANDOMIZE TIMER()
FOR col = column TO column + 80
MAKE OBJECT BOX col,20,99.8,20
TEXTURE OBJECT col,wall
SCALE OBJECT TEXTURE col,5,30
POSITION OBJECT col,RND(515)-20,49.95, RND(270)+710
NEXT col
ENDFUNCTION
```

continued on next page

LISTING-32.15

(continued)

Drawing the Castle

```
FUNCTION DrawTrees()  
  REM *** Create trees ***  
  FOR c = tree1 TO tree1 + 30  
    MAKE OBJECT PLAIN c,25,35  
    TEXTURE OBJECT c, tree  
    SET OBJECT TRANSPARENCY c,1  
    POSITION OBJECT c,-45,17,(c-6) * 17.5  
  NEXT c  
ENDFUNCTION
```

The function *DrawInternalColumns()* adds randomly placed columns within the castle. This will allow our player to have obstacles to navigate without having to go to a great deal of trouble designing an exact layout for the castle's interior.

The *DrawTrees()* function draws a set of trees by texturing a set of planes with a tree image.

Activity 32.27

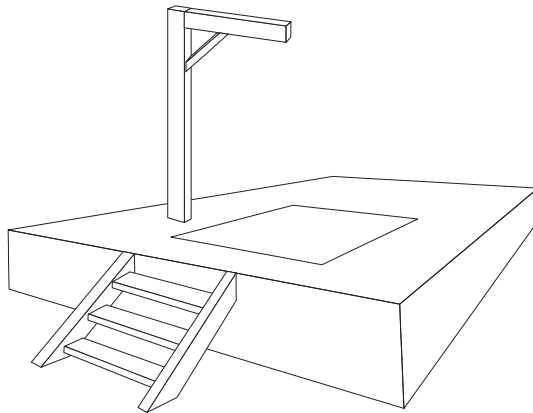
Type in and test the program given above (*castle01.dbpro*).

Activity 32.28

Remove the main section from the program, leaving only the constants and functions. Save this as *castle.dbpro*.

Activity 32.29

Create a program (*gallows.dbpro*) containing a function, *DrawGallows()*, that produces a 3D gallows similar to that in the sketch below. Use any appropriate textures.



The gallows platform should be centred on (-125,7.5,0) and be 15 units high, and 50 units in width and depth. Place the gallows on a 300 by 300 cobbled plane.

In the main section of the program include the lines

```
POSITION CAMERA 0,8,-100  
POINT CAMERA -150,10,0
```

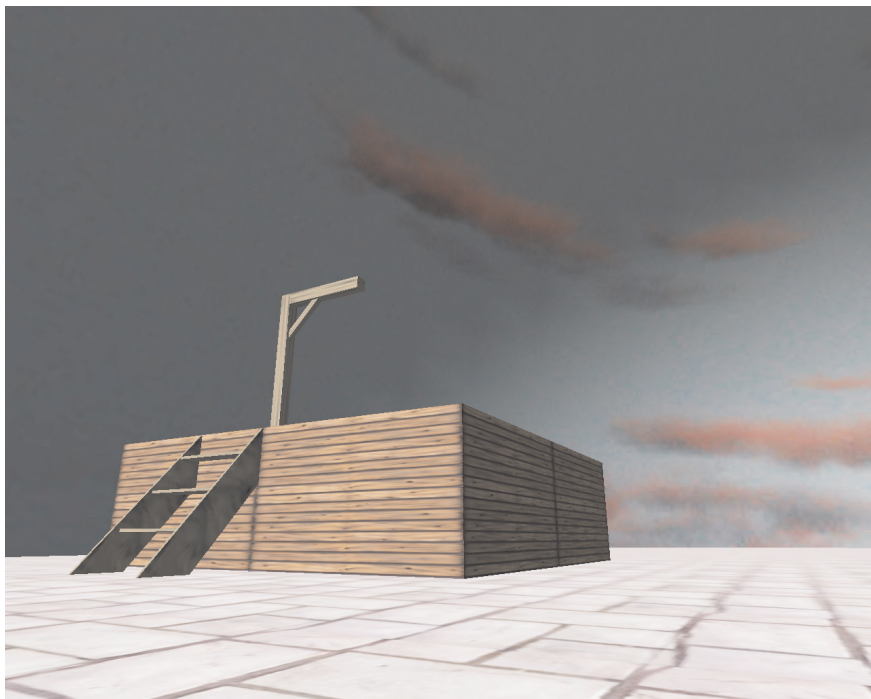
after the call to the *DrawGallows()* function. This will ensure that the camera is pointing at the gallows.

Sky Spheres

Both the castle and the gallows look a bit out of place with the blue background. One way to create a more natural environment is to place a large sphere round the whole thing and texture that sphere with an image of the sky. This is known as a **sky sphere**. FIG-32.32 shows the results obtained by adding a sky sphere to the gallows program.

FIG-32.32

Using a Sky Sphere



To implement a sky sphere in our gallows program, we'll start by making the ground plane set up in *DrawGallows()* a bit larger:

```
MAKE OBJECT PLAIN GroundObj, 2000, 2000
```

and then increase the tiling so the cobbles don't get too large:

```
SCALE OBJECT TEXTURE GroundObj, 150, 150
```

These are the only changes required in the *DrawGallows()* function. Now we can add a few lines to the main section. First we need the image to be used to texture the sphere with a sky effect:

```
LOAD IMAGE "sky.jpg", 1
```

Next we can create the sphere with the same diameter as the plane:

```
MAKE OBJECT SPHERE 1, -2000, 50, 50
```

Notice that the sphere has been created with extra polygons. This helps smooth out the sky background.

Apparently all we need to do now is texture the sphere:

```
TEXTURE OBJECT 1, 1
```

Activity 32.30

Modify your *gallows.dbpro* program using the lines given above. The main section should be coded as:

```
REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32

DrawGallows()

REM *** Create the sky sphere ***
LOAD IMAGE "sky.jpg",1
MAKE OBJECT SPHERE 1,2000,50,50
TEXTURE OBJECT 1,1

REM *** Set up camera ***
POSITION CAMERA 0,8,-100
POINT CAMERA -150,10,0

REM *** End program ***
WAIT KEY
END
```

Don't worry if you don't see any sky!

We can't see the texture on the sphere because we're on the inside of the sphere and DarkBASIC Pro has culled the polygons that make up the sphere.

There are two ways to solve this problem. The first is to switch off culling on the sphere. This can be done using the line:

```
SET OBJECT CULL 1,0
```

Activity 32.31

Add the above line to the main section of your code. Is the sky now visible?

An alternative way of displaying the sphere's texture is to turn the sphere inside out! This is done by specifying a negative size for the sphere when it is being created.

Activity 32.32

Change the line

```
MAKE OBJECT SPHERE 1,2000,50,50
```

to have a negative size value:

```
MAKE OBJECT SPHERE 1,-2000,50,50
```

and remove the SET OBJECT CULL statement.

How does this affect the sphere's texture?

To cure the problem of an inverted mirror image (which isn't actually a problem in this case) we need to save the original image as an inverted mirror image in the first place and this will then be reversed when the image is used as a texture.

Activity 32.33

If you have an appropriate paint package, invert and mirror the image *sky.jpg* (save the resulting image as *skyIM.jpg*) and use the new image as a texture for the sky sphere.

If you don't have an appropriate package, the inverted mirror image is supplied with the images for this chapter.

Summary

- A sky sphere allows us to create a sky affect around our 3D world.
- A sky sphere is a large sphere textured with an image of the sky.
- To make the sphere's texture visible from within the sphere, switch off the sphere's culling or create and inverted mirror image of the sky and use it to texture a sphere with a negative diameter value.

Solutions

Activity 32.1

```
REM *** Set display resolution ***
SET DISPLAY MODE 1280,1024,32
REM *** Load texture image ***
LOAD IMAGE "eyecol.bmp",1
REM *** Create cube ***
MAKE OBJECT CUBE 1, 40
REM *** Add texture to cube ***
TEXTURE OBJECT 1,1
REM *** Position cube ***
POSITION OBJECT 1,25,0,100
REM *** Rotate cube continuously ***
DO
    TURN OBJECT LEFT 1, 1.0
LOOP
REM *** End program ***
END
```

Activity 32.2

To change shape requires the line

```
MAKE OBJECT CUBE 1, 40
```

to be replaced by each of the following in turn:

```
MAKE OBJECT BOX 1, 10,20,30
MAKE OBJECT CYLINDER 1, 15
MAKE OBJECT CONE 1, 15
MAKE OBJECT SPHERE 1, 10
```

The cube and box repeat the texture image on each side; other shapes show the image only once.

Activity 32.3

No solution required.

Activity 32.4

The texture seems a little more blurred when using mipmaps, but the texturing process seems to be carried out at a faster frame rate.

Activity 32.5

No solution required.

Activity 32.6

No solution required.

Activity 32.7

```
REM *** Set display resolution ***
SET DISPLAY MODE 1280,1024,32
REM *** Load texture image ***
LOAD IMAGE "seamlesseye.bmp",1
REM *** Create and texture sphere ***
MAKE OBJECT SPHERE 1, 40,40,40
TEXTURE OBJECT 1,1
DO
    TURN OBJECT LEFT 1, 1.0
LOOP
END
```

Activity 32.8

By changing the coordinates of the bottom right corner of the video to 10,10, the video itself is only 11 pixels by 11 pixels, and, when expanded to cover the surface of the cube becomes indistinct and blocky.

By changing the corner values from 10,10 to 1000,1000 we make the video 1001 pixels by 1001 pixels (actually larger than the original recording). Since we cannot add any detail which was not in the original recording, this size does not achieve any better results than a lower resolution (say 640 by 640), but does increase the load on the video hardware and slows down the whole process.

Activity 32.9

No solution required.

Activity 32.10

No solution required.

Activity 32.11

```
REM *** Set display resolution ***
SET DISPLAY MODE 1280,1024,32
REM *** Load image ***
LOAD IMAGE "eyecol.bmp",1
REM *** Create and texture plain object ***
MAKE OBJECT PLAIN 1,200,200
TEXTURE OBJECT 1,1
REM *** Offset texture on image***
SCROLL OBJECT TEXTURE 1,0.1,0.0
WAIT KEY
SCROLL OBJECT TEXTURE 1,0.1,0.0
REM *** End program ***
WAIT KEY
END
```

We can see from the results produced by the program that the scroll effect is cumulative, with the image moving another step to the left when the second SCROLL OBJECT TEXTURE statement is applied.

Activity 32.12

```
REM *** Set display resolution ***
SET DISPLAY MODE 1280,1024,32
REM *** Load image ***
LOAD IMAGE "eyecol.bmp",1
REM *** Create and texture plain object ***
MAKE OBJECT PLAIN 1,200,200
TEXTURE OBJECT 1,1
REM *** Offset texture placed on image***
DO
    SCROLL OBJECT TEXTURE 1,0.1,0.0
LOOP
REM *** End program ***
WAIT KEY
END
```

The image scrolls vertically.

```
REM *** Set display resolution ***
SET DISPLAY MODE 1280,1024,32
REM *** Load image ***
LOAD IMAGE "eyecol.bmp",1
```

```

REM *** Create and texture plain object ***
MAKE OBJECT PLAIN 1,200,200
TEXTURE OBJECT 1,1
REM *** Offset texture placed on image***
DO
    SCROLL OBJECT TEXTURE 1,0.0,0.1
LOOP
REM *** End program ***
WAIT KEY
END

```

Activity 32.13

```

REM *** Set display resolution ***
SET DISPLAY MODE 1280,1024,32
REM *** Make and position spheres ***
LOAD IMAGE "eyecol.bmp",1
MAKE OBJECT SPHERE 1, 40
POSITION OBJECT 1,25,-20,100
MAKE OBJECT SPHERE 2, 40
POSITION OBJECT 2, -25,-20,100
REM *** Texture spheres ***
TEXTURE OBJECT 1,1
TEXTURE OBJECT 2,1
REM *** Offset so eyes are at front ***
SCROLL OBJECT TEXTURE 1, 0.51,0.0
SCROLL OBJECT TEXTURE 2,0.51,0.0
REM *** Make eyes follow mouse ***
DO
    x3D = MOUSEX() - SCREEN WIDTH()/2
    y3D = -(MOUSEY() - SCREEN HEIGHT()/2)
    POINT OBJECT 1, x3D,y3D,-300
    POINT OBJECT 2, x3D,y3D,-300
LOOP
REM *** End program ***
WAIT KEY
END

```

Activity 32.14

No solution required.

Activity 32.15

```

REM *** Set display resolution ***
SET DISPLAY MODE 1280,1024,32
REM *** Load texture image ***
LOAD IMAGE "DoNot.bmp",2
REM *** Make and position cube ***
MAKE OBJECT CUBE 1, 40
POSITION OBJECT 1,25,0,100
REM *** Texture cube with image ***
TEXTURE OBJECT 1,2
REM *** Make black areas of texture ***
REM *** transparent ***
SET OBJECT TRANSPARENCY 1,1
REM *** Rotate cube ***
DO
    PITCH OBJECT DOWN 1, 1.0
    TURN OBJECT LEFT 1, 1.0
LOOP
REM *** End program ***
END

```

Any part of the cube which is textured with black disappears.

Activity 32.16

```

REM *** Set display resolution ***
SET DISPLAY MODE 1280,1024,32
REM *** Make magenta transparent ***
SET IMAGE COLORKEY 255,0,255
REM *** Load texture images ***

```

```

LOAD IMAGE "textureWood.jpg",1
LOAD IMAGE "DoNotMag.bmp",2
REM *** Create and texture cube ***

MAKE OBJECT CUBE 1, 40
TEXTURE OBJECT 1,1
REM *** Add secondary texture ***
SET DETAIL MAPPING ON 1,2
REM *** Position cube ***
POSITION OBJECT 1,25,0,100
REM *** Rotate cube ***
DO PITCH OBJECT DOWN 1, 1.0
    TURN OBJECT LEFT 1, 1.0
LOOP
REM *** End program ***
END

```

Activity 32.17

```

REM *** Set display resolution ***
SET DISPLAY MODE 1280,1024,32
REM *** Set magenta as transparent ***
SET IMAGE COLORKEY 255,0,255
REM *** Load texture images ***
LOAD IMAGE "textureWood.jpg",1
LOAD IMAGE "DoNotMag.bmp",2
LOAD IMAGE "FlagMag.bmp",3
REM *** Create and texture cube ***
MAKE OBJECT CUBE 1, 40
TEXTURE OBJECT 1,1
REM *** Add text as secondary texture ***
SET DETAIL MAPPING ON 1,2
REM *** Try using another texture ***
SET DETAIL MAPPING ON 1,3
REM *** Position cube ***
POSITION OBJECT 1,25,0,100
REM *** Rotate cube ***
DO PITCH OBJECT DOWN 1, 1.0
    TURN OBJECT LEFT 1, 1.0
LOOP
REM *** End program ***
END

```

Only the wood and flag textures show; the text *"DO NOT OPEN"* is missing.

Activity 32.18

```

REM *** Set display resolution ***
SET DISPLAY MODE 1280,1024,32
REM *** Set magenta as transparent ***
SET IMAGE COLORKEY 255,0,255
REM *** Load texture images ***
LOAD IMAGE "textureWood.jpg",1
LOAD IMAGE "DoNotMag.bmp",2
REM *** Create and texture cube ***
MAKE OBJECT CUBE 1, 40
TEXTURE OBJECT 1,1
REM *** Tile cube's texture ***
SCALE OBJECT TEXTURE 1,2,2
REM *** Add secondary texture ***
SET DETAIL MAPPING ON 1,2
REM *** Position cube ***
POSITION OBJECT 1,25,0,100
REM *** Rotate cube ***
DO PITCH OBJECT DOWN 1, 1.0
    TURN OBJECT LEFT 1, 1.0
LOOP
REM *** End program ***
END

```

The DETAIL MAPPING image is also tiled.

Activity 32.19

No solution required.

```
WAIT 10
LOOP
REM *** End program ***
END
```

Activity 32.20

No solution required.

The cube fades until it is completely invisible.

Activity 32.21

```
REM *** Set display resolution ***
SET DISPLAY MODE 1280,1024,32
REM *** Seed random number generator ***
RANDOMIZE TIMER()
REM *** Load image ***
LOAD IMAGE "DoNot.bmp",1
REM *** Make and position cube ***
MAKE OBJECT CUBE 1, 40
REM *** Create detail mapping ***
SET DETAIL MAPPING ON 1,1
POSITION OBJECT 1,25,0,100
REM *** Colour cube red ***
COLOR OBJECT 1, RGB(255,0,0)
REM *** Rotate Cube ***
DO
    PITCH OBJECT DOWN 1,1.0
    TURN OBJECT LEFT 1, 1.0
    REM *** 1 in 1000 of going green ***
    IF RND(1000) = 500
        COLOR OBJECT 1, RGB(0,255,0)
    ENDIF
LOOP
REM *** End program ***
END
```

Activity 32.22

No solution required.

Activity 32.23

No solution required.

Activity 32.24

```
REM *** Set display resolution ***
SET DISPLAY MODE 1280,1024,32
REM *** Load texture images ***
LOAD IMAGE "textureWood.jpg",1
LOAD IMAGE "DoNot.bmp",2
REM *** Make and position cube ***
MAKE OBJECT CUBE 1, 40
POSITION OBJECT 1,25,0,100
REM *** Texture cube ***
TEXTURE OBJECT 1,1
REM *** Create background plain ***
MAKE OBJECT PLAIN 2,100,100
TEXTURE OBJECT 2, 2
POSITION OBJECT 2,0,0,200
REM *** Start reflective value at 200 ***
reflectivity = 200
REM *** Make cube transparent ***
GHOST OBJECT ON 1,0
REM *** rotate cube ***
DO
    PITCH OBJECT DOWN 1, 1.0
    TURN OBJECT LEFT 1, 1.0
    FADE OBJECT 1,reflectivity
    REM *** Reduce reflectivity to zero ***
    IF reflectivity > 0
        DEC reflectivity
    ENDIF
END
```

Activity 32.25

No solution required.

Activity 32.26

```
REM *** Set display resolution ***
SET DISPLAY MODE 1280, 1024,32
REM *** Create cube ***
MAKE OBJECT CUBE 1,10
REM *** Texture cube ***
LOAD IMAGE "windmillshaped.tga",1
TEXTURE OBJECT 1,1
REM *** Rotate cube ***
DO
    REM *** IF key pressed, ghost ***
    IF INKEY$() <> ""
        GHOST OBJECT ON 1,3
    ENDIF
    TURN OBJECT LEFT 1,1
LOOP
REM *** End program ***
END
```

Activity 32.27

No solution required.

Activity 32.28

No solution required.

Activity 32.29

```
REM *** Set display resolution ***
SET DISPLAY MODE 1280,1024,32
DrawGallows()
POINT CAMERA -150,10,0
WAIT KEY
END

FUNCTION DrawGallows()
    REM *** Set up names ***
    REM *** Object names ***
    #CONSTANT GroundObj 901
    #CONSTANT PlatformObj 902
    #CONSTANT VerticalPostObj 903
    #CONSTANT HorizontalPostObj 904
    #CONSTANT DiagonalPostObj 905
    #CONSTANT TopStepObj 906
    #CONSTANT MiddleStepObj 907
    #CONSTANT BottomStepObj 908
    #CONSTANT StepEdgeRightObj 909
    #CONSTANT StepEdgeLeftObj 910
    REM *** Image names ***
    #CONSTANT CobbleImg 901
    #CONSTANT PlanksImg 902
    #CONSTANT WoodImg 903
    REM *** Load texture images ***
    LOAD IMAGE "TextureWood.jpg",PlanksImg
    LOAD IMAGE "CobbleStones.jpg",CobbleImg
    LOAD IMAGE "Wood.jpg",WoodImg
    REM *** Create cobbled square ***
    MAKE OBJECT PLAIN GroundObj,300,300
```

```

TEXTURE OBJECT GroundObj,CobbleImg
SCALE OBJECT TEXTURE GroundObj,30,30
XROTATE OBJECT GroundObj,-90
REM *** Create platform ***
MAKE OBJECT BOX PlatformObj,50,15,50
TEXTURE OBJECT PlatformObj,PlanksImg
SCALE OBJECT TEXTURE PlatformObj,2,2
POSITION OBJECT PlatformObj,-125,7.5,0
REM *** Create vertical post ***
MAKE OBJECT BOX VerticalPostObj,2,30,2
TEXTURE OBJECT VerticalPostObj,WoodImg
POSITION OBJECT VerticalPostObj,
    ↵-147.5,30,0
REM *** Create horizontal post ***
MAKE OBJECT BOX HorizontalPostObj,
    ↵2,15,2
TEXTURE OBJECT HorizontalPostObj,
    ↵WoodImg
ZROTATE OBJECT HorizontalPostObj,90
POSITION OBJECT HorizontalPostObj,
    ↵-139,44,0
REM *** Create diagonal post ***
MAKE OBJECT BOX DiagonalPostObj,1,10,1
TEXTURE OBJECT DiagonalPostObj,WoodImg
ZROTATE OBJECT DiagonalPostObj,-45
POSITION OBJECT DiagonalPostObj,
    ↵-144,40,0
REM *** Make top step ***
MAKE OBJECT BOX TopStepObj,10,0.3,3
TEXTURE OBJECT TopStepObj,WoodImg
POSITION OBJECT TopStepObj,-130,12,-26
REM *** Make middle step ***
CLONE OBJECT MiddleStepObj,TopStepObj
POSITION OBJECT MiddleStepObj,-130,8,-29
REM *** Make bottom step ***
CLONE OBJECT BottomStepObj,TopStepObj
POSITION OBJECT BottomStepObj,-130,4,-32
REM *** Make right step edge***
MAKE OBJECT BOX StepEdgeRightObj,0.3,4,20
TEXTURE OBJECT StepEdgeRightObj,WoodImg
XROTATE OBJECT StepEdgeRightObj,-50
POSITION OBJECT StepEdgeRightObj,
    ↵-125,6,-30
REM *** Make left step edge ***
CLONE OBJECT StepEdgeLeftObj,
    ↵StepEdgeRightObj
POSITION OBJECT StepEdgeLeftObj,
    ↵-135,6,-30
ENDFUNCTION

```

Activity 32.30

The changes to the cobbled square in the *DrawGallows()* function are shown in bold below:

```

REM *** Create cobbled square ***
MAKE OBJECT PLAIN GroundObj,2000,2000
TEXTURE OBJECT GroundObj,CobbleImg
SCALE OBJECT TEXTURE GroundObj,150,150

```

Activity 32.31

The sky should now be visible.

Activity 32.32

The sphere's texture is upside down and mirrored.

Activity 32.33

No solution required.

Changing the Camera's Field of View
Controlling Camera Movement with the Keyboard
Controlling Camera Movement with the Mouse
Creating and Deleting Extra Cameras
Linking the Camera to a 3D Object
Pointing the Camera
Positioning the Camera
Rotating the Camera
Setting the Camera's Output to the Screen
Specifying the Distance Limits of Camera Vision

Camera Basics

Introduction

Although we've seen how to move objects about in 3D space, our viewpoint has remained fixed. That viewpoint is determined by the position of a camera. We can think of what we see on the screen as a picture being broadcast by a TV camera; if the camera moves then the picture we see on the screen shifts as well.

The camera in DarkBASIC Pro is a virtual one; totally invisible and with no chance of accidentally appearing in the picture showing on the screen.

In this chapter we'll cover the DarkBASIC Pro statements that allow us to move the main camera, create new cameras (so we can view the world from multiple positions) and take full control of the view created on our screen.

We'll use the castle environment we created in the previous chapter to demonstrate most of the camera statements.

Positioning the Camera

Any program containing a 3D object automatically contains a camera. It is this camera (camera 0) whose "transmitted" image appears on the screen.

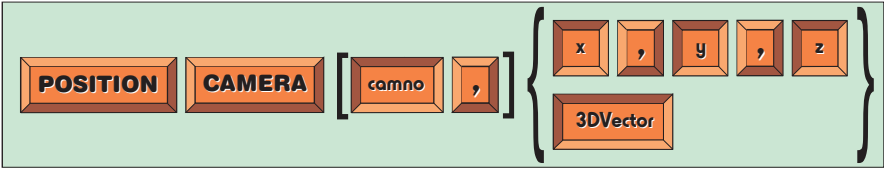
Just as we might position, move, or rotate a 3D object, so we can perform the same type of operation on the camera. By moving the camera and making it point in some other direction, so we generate a new viewpoint on the screen.

The POSITION CAMERA Statement

We can position the camera at any point in 3D space using the POSITION CAMERA statement which has the format shown in FIG-33.1.

FIG-33.1

The POSITION
CAMERA Statement



In the diagram:

- | | |
|-----------------|--|
| <i>camno</i> | is an integer value specifying which camera is to be moved. If no camera number is given then the currently active camera is used. |
| <i>x,y,z</i> | is a set of real values giving the position to which the camera is to be moved. |
| <i>3Dvector</i> | is an integer value giving the ID of the vector containing the coordinates to which the camera is to be moved. |

Only camera zero is created automatically. If you don't intend to create any others,

then there is no need to specify a camera number in most camera-related instructions.

The position to which the camera is to be moved can be given using three separate values or a single 3-element vector object containing the same three values. We'll ignore the vector option until we cover 3D vectors in a later chapter. So, using the POSITION CAMERA statement we could place the main camera at position (10,20,30) using the line:

```
POSITION CAMERA 0,10,20,30
```

Activity 33.1

Create a new program (*camera01.dbpro*) and include the code in *castle.dba* (copy *castle.dba* to this project's folder and use #INCLUDE).

In the main section call *DrawCastle()*, followed by the lines:

```
WAIT KEY
POSITION CAMERA 0,0,5,50
WAIT KEY
END
```

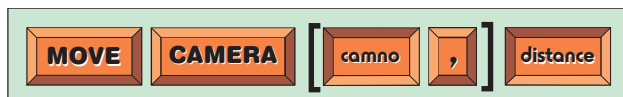
How does this affect the view when the program is executed?

The MOVE CAMERA Statement

If we want the camera to travel in the direction in which it is currently facing, then we can use the MOVE CAMERA statement. This statement has the format shown in FIG-33.2.

FIG-33.2

The MOVE CAMERA Statement



In the diagram:

camno

is an integer value specifying which camera is to be moved. If no camera number is given then the currently active camera is used.

distance

is a real number specifying the number of units the camera is to be moved. The camera moves in the direction it is currently pointing. If a negative value is given, the camera will move backwards.

For example, we could move the active camera forward 30 units using the line:

```
MOVE CAMERA 30
```

Activity 33.2

Replace the POSITION CAMERA statement in the last program with the line

```
MOVE CAMERA 50
```

How does this affect the view produced?

As a result of the last Activity the camera, which was facing slightly downwards, has passed through the plane objects that make up the pathway and grass. We can see this happen if we move the camera just one unit at a time.

Activity 33.3

Replace the MOVE CAMERA statement with the following lines:

```
FOR c = 1 TO 100
  MOVE CAMERA 1
  WAIT 50
NEXT c
```

Run the program and observe how the camera moves under the plane.

Changing the Viewpoint

The POINT CAMERA Statement

It's all very well to move a camera, but we also need to point the camera in the desired direction. One way to do this is to use the POINT CAMERA statement which has the format shown in FIG-33.3.

FIG-33.3

The POINT CAMERA Statement



In the diagram:

camno

is an integer value specifying which camera is to be moved. If no camera number is given then the currently active camera is used.

x,y,z

is a set of real values specifying the position in space where the camera is to be aimed. The point specified will appear in the middle of the screen.

For example, we could point the camera at the entrance to the castle using the line:

```
POINT CAMERA -25,5,700
```

Activity 33.4

Add the above statement to your last program immediately before the FOR loop used to move the camera.

Extend the FOR loop so that the camera moves right up to the castle's wall.

The ROTATE CAMERA Statement

The camera can be rotated to an exact angle about any of its own local axes (see FIG-33.4) using the ROTATE CAMERA statement.

FIG-33.4

Absolute Camera
Rotation

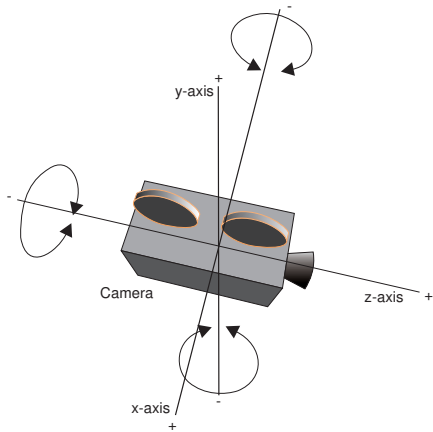
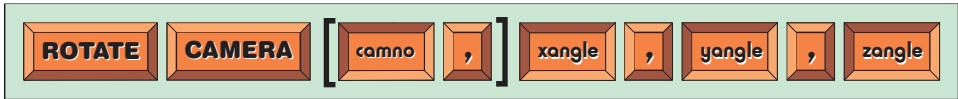


FIG-33.5

The ROTATE
CAMERA
Statement



In the diagram:

- camno* is the integer value specifying the camera to be rotated.
- xangle* is a real number giving the angle (in degrees) to which the camera is to be rotated about its x-axis.
- yangle* is a real number giving the angle (in degrees) to which the camera is to be rotated about its y-axis.
- zangle* is a real number giving the angle (in degrees) to which the camera is to be rotated about its z-axis.

Rotation about the x-axis is the type of movement we would make filming a rocket taking off or a diver jumping off a cliff. Rotation about the y-axis (also known as panning) is the type of movement required to follow a racing car or a runner.

To make the camera point straight forward (that is, parallel to the main z-axis) we would use the line:

```
ROTATE CAMERA 0,0,0
```

Activity 33.5

Remove the POINT CAMERA and FOR loop code from *camera01.dbpro* and add the lines

```
FOR angle = -45 TO 45
  ROTATE CAMERA angle,0,0
  WAIT 10
NEXT angle
```

Run the program and observe the effect produced. Change the code to also create rotation about the other two axes.

The SET CAMERA ROTATION Statement

When the camera is required to rotate about all three axes at the same time, it normally performs rotation about the z-axis first, y-axis second and x-axis last. Of course, this will be performed so quickly that you'll probably not be aware of the order of events. However, if, for some reason we need to reverse the order of rotations (x first, y second, and z last), we can use the SET CAMERA ROTATION statement. Once set, the order in which the axes are handled will remain on this new setting unless you revert to normal using a second option of the SET CAMERA ROTATION statement. The statement has the format shown in FIG-33.6.

FIG-33.6

The SET CAMERA ROTATION Statement



In the diagram:

- XYZ
- Use this option to set rotation order to x-axis, y-axis, and, finally, z-axis.
- ZYX
- Use this option to revert to the default rotation order z-axis, y-axis, x-axis.

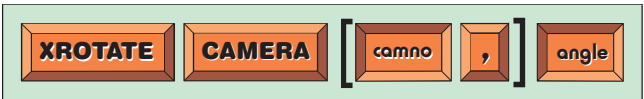
Rather than perform three rotations in a single statement, we can rotate about a single axis only using one of the following:

The XROTATE CAMERA Statement

To rotate the camera about the x-axis we use the XROTATE OBJECT statement which has the format shown in FIG-33.7.

FIG-33.7

The XROTATE CAMERA Statement



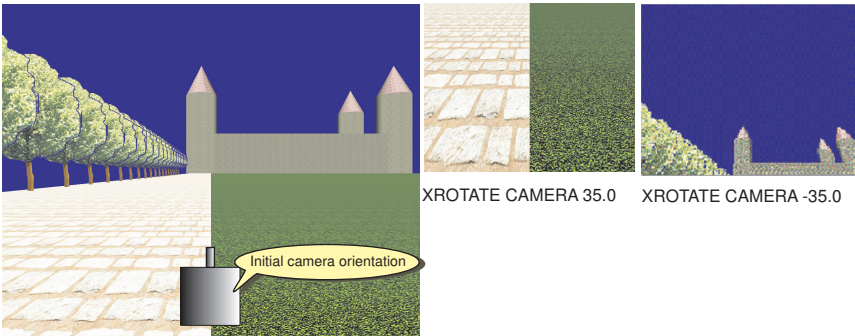
In the diagram:

- camno
- is an integer value giving the ID of the camera to be rotated.
- angle
- is a real number giving the angle (in degrees) to which the object is to be rotated.

FIG-33.8 shows the result of rotating the active camera to 35.0° and -35.0° about the x-axis.

FIG-33.8

The Effects of the XROTATE Statement



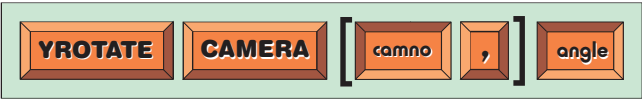
Initial Camera View

The YROTATE CAMERA Statement

To rotate the camera about the y-axis we use the YROTATE OBJECT statement which has the format shown in FIG-33.9.

FIG-33.9

The YROTATE
CAMERA Statement



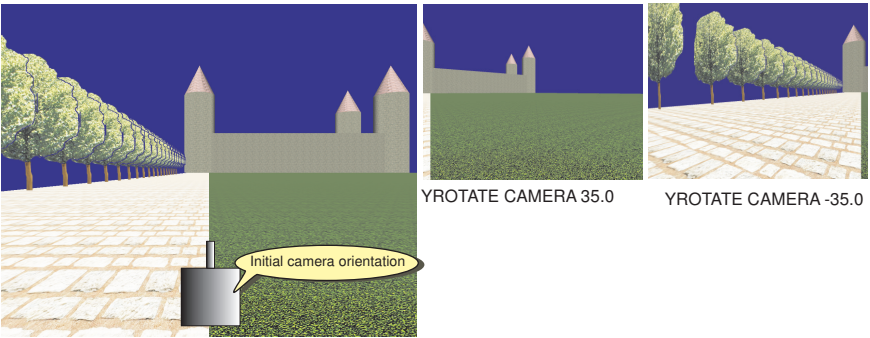
In the diagram:

- camno* is an integer value giving the ID of the camera to be rotated.
- angle* is a real number giving the angle (in degrees) to which the object is to be rotated.

FIG-33.10 shows the result of rotating the active camera to 35.0° and -35.0° about the y-axis.

FIG-33.10

The Effects of the
YROTATE
CAMERA Statement



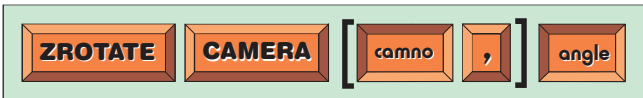
Initial Camera View

The ZROTATE CAMERA Statement

To rotate the camera about the z-axis we use the ZROTATE OBJECT statement which has the format shown in FIG-33.11.

FIG-33.11

The ZROTATE
CAMERA Statement



In the diagram:

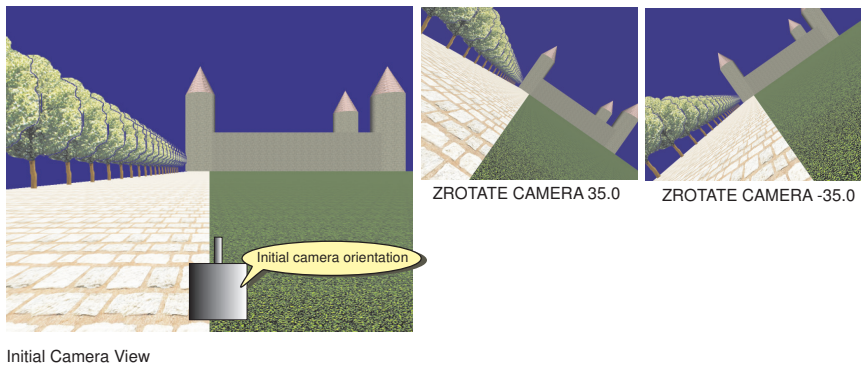
- camno* is an integer value giving the ID of the camera to be rotated.
- angle* is a real number giving the angle (in degrees) to which the object is to be rotated.

FIG-33.12 shows the result of rotating the active camera to 35.0° and -35.0° about the z-axis.

We might require such a rotation if we wanted to show the view from the inside as a car rolls over onto its roof.

FIG-33.12

The Effects of the
ZROTATE CAMERA
Statement



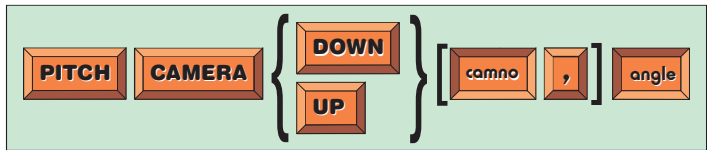
The following statements allow relative rotation rather than absolute rotation.

The PITCH CAMERA Statement

We can tilt the camera upwards (i.e. rotate it in a positive direction about the x-axis) or downwards (rotating in a negative direction about the x-axis) using the PITCH CAMERA statement which has the format shown in FIG-33.13.

FIG-33.13

The PITCH CAMERA
Statement



In the diagram:

DOWN, UP

Choose DOWN to make the camera tilt downwards;
choose UP to make the camera tilt upwards.

camno

is the integer value representing the camera to be
rotated.

angle

is a real number giving the number of degrees to
be added to the camera's current rotation about its
x-axis.

PITCH CAMERA UP moves the camera in the same direction as XROTATE CAMERA using a positive angle.

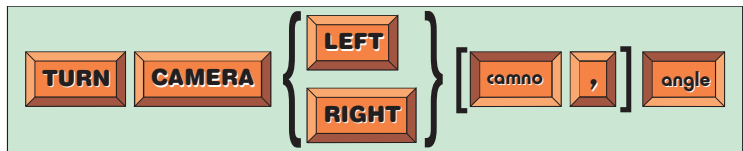
PITCH CAMERA DOWN moves the camera in the same direction as XROTATE CAMERA using a negative angle.

The TURN CAMERA Statement

We can turn the camera right (i.e. rotate it in a positive direction about the y-axis) or left (rotating in a negative direction about the y-axis) using the TURN CAMERA statement which has the format shown in FIG-33.14.

FIG-33.14

The TURN CAMERA
Statement



In the diagram:

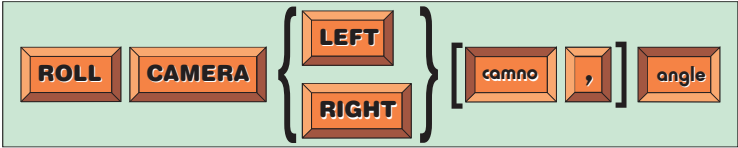
LEFT, RIGHT	Choose LEFT to make the camera rotate to the left; choose RIGHT to make the camera rotate the right.
<i>camno</i>	is the integer value representing the camera to be rotated.
<i>angle</i>	is a real number giving the number of degrees to be added to the camera's current rotation about its y-axis.

TURN CAMERA RIGHT moves the camera in the same direction as YROTATE CAMERA using a positive angle. TURN CAMERA LEFT moves the camera in the same direction as YROTATE CAMERA using a negative angle.

The ROLL CAMERA Statement

We can roll the camera to the left (i.e. rotate it in a positive direction about the z-axis) or right (rotating in a negative direction about the z-axis) using the ROLL CAMERA statement which has the format shown in FIG-33.15.

FIG-33.15
The ROLL CAMERA Statement



In the diagram:

LEFT, RIGHT	Choose LEFT to make the camera roll to the left; choose RIGHT to make the camera roll to the right.
<i>camno</i>	is the integer value representing the camera to be rotated.
<i>angle</i>	is a real number giving the number of degrees to be added to the camera's current rotation about its z-axis.

ROLL CAMERA RIGHT moves the camera in the same direction as ZROTATE CAMERA using a positive angle. ROLL CAMERA LEFT moves the camera in the same direction as ZROTATE CAMERA using a negative angle.

Activity 33.6

Create a new main section for *camera01.dbpro*, so that the camera can be rotated using the keyboard. Each rotation should be by 1° in the required direction. Start the camera with zero rotation. Control keys are as follows:

Key	Direction	Key	Direction
z	left y-axis	/	down x-axis
x	right y-axis	n	left z-axis
'	up x-axis	m	right z-axis

Retrieving Camera Data

Statements are available to retrieve the position and orientation of any camera. These are explained below.

The CAMERA POSITION Statement

The CAMERA POSITION statement has three options, each allowing a single ordinate of a specified camera's position to be discovered. The CAMERA POSITION statement has the format shown in FIG-33.16.

FIG-33.16

The CAMERA POSITION Statement



In the diagram:

- X

Use this option to return the x-ordinate of the camera's current position.
- Y

Use this option to return the y-ordinate of the camera's current position.
- Z

Use this option to return the z-ordinate of the camera's current position.
- camno*

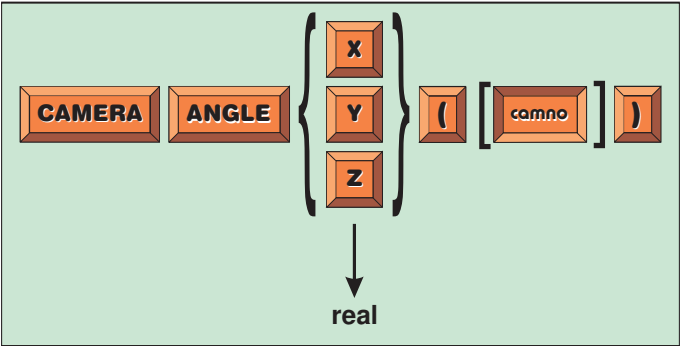
is an integer value specifying the camera whose position is required. If this option is omitted the currently active camera is assumed.

The CAMERA ANGLE Statement

The angle to which a camera has been rotated about any of the local axes can be determined using the CAMERA ANGLE statement which has the format shown in FIG-33.17.

FIG-33.17

The CAMERA ANGLE Statement



In the diagram:

X	Use this option to return the camera's current angle of rotation about its x-axis.
Y	Use this option to return the camera's current angle of rotation about its y-axis.
Z	Use this option to return the camera's current angle of rotation about its z-axis.
<i>camno</i>	is an integer value specifying the camera whose position is required. If this option is omitted the currently active camera is assumed.

Activity 33.7

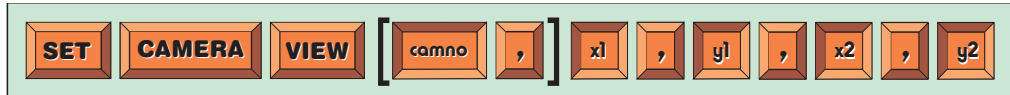
Modify your previous program so that the camera's angle of rotation about all three local axes is always displayed on the screen.

Modifying Camera Characteristics

The SET CAMERA VIEW Statement

Normally, the picture created by our imaginary camera occupies the full screen, but we can change this using the SET CAMERA VIEW statement which allows us to specify a rectangular area of the screen where camera output should appear. The SET CAMERA VIEW statement has the format shown in FIG-33.18.

FIG-33.18 The SET CAMERA VIEW Statement



In the diagram:

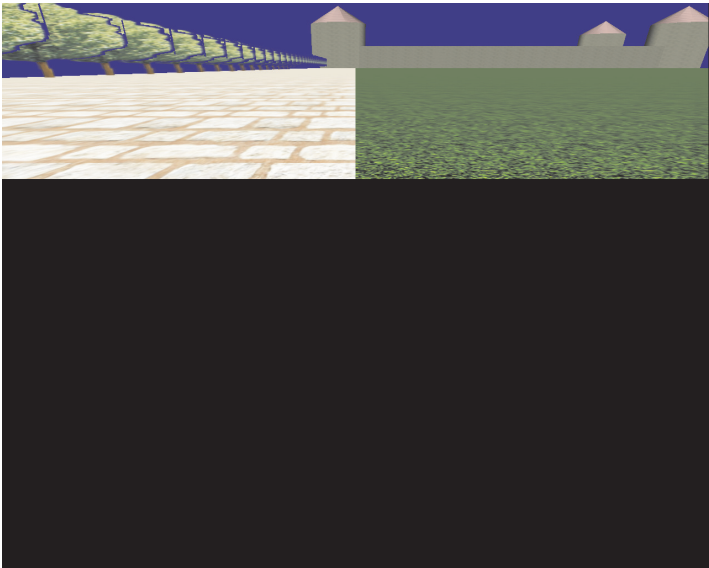
<i>camno</i>	is an integer value specifying the camera whose output area on the screen is to be defined. If this option is omitted the currently active camera is assumed.
<i>x1,y1,x2,y2</i>	This is a set of integer values specifying the coordinates of the top-left (<i>x1,y1</i>) and bottom-right (<i>x2,y2</i>) corners of the rectangular area of the screen to be used as camera output.

For example, we could change the output to occupy the area shown in FIG-33.19 using the statement:

```
SET CAMERA VIEW 0,0,1279,319
```

FIG-33.19

Changing the Area of the Screen Occupied by the Camera's Output



Activity 33.8

Modify your last program to use the area specified in the line above for camera output.

Move the camera angle displays into the black area of the screen. Does this cause any problems?

The SET CAMERA ASPECT Statement

If we use SET CAMERA VIEW to change the ratio of the width-to-height of the picture, we'll end up with a distorted image, as we saw in FIG-33.19. However, we can correct this using the SET CAMERA ASPECT statement which allows us to define width-to-height aspect ratio. In effect, by making this ratio the same as that created by the SET CAMERA VIEW we can remove any distortion. In the SET CAMERA VIEW example on the previous page we set the area containing the camera output to (0,0), (1279,319). This gives a width of 1280 and a height of 320 giving a width-to-height ratio of 4-to-1.

The SET CAMERA ASPECT statement has the format shown in FIG-33.20.

FIG-33.20

The SET CAMERA ASPECT Statement



In the diagram:

- camno*

is an integer value specifying the camera whose aspect ratio is to be defined. If this option is omitted, the currently active camera is assumed.
- ratio*

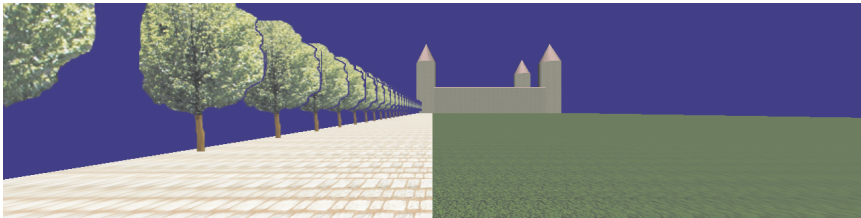
is a real number giving the aspect ratio of width to height.

To correct the distortion caused by the SET CAMERA VIEW statement we need the line

giving the result shown in FIG-33.21.

FIG-33.21

Correcting the Camera
Output Distortion



Activity 33.9

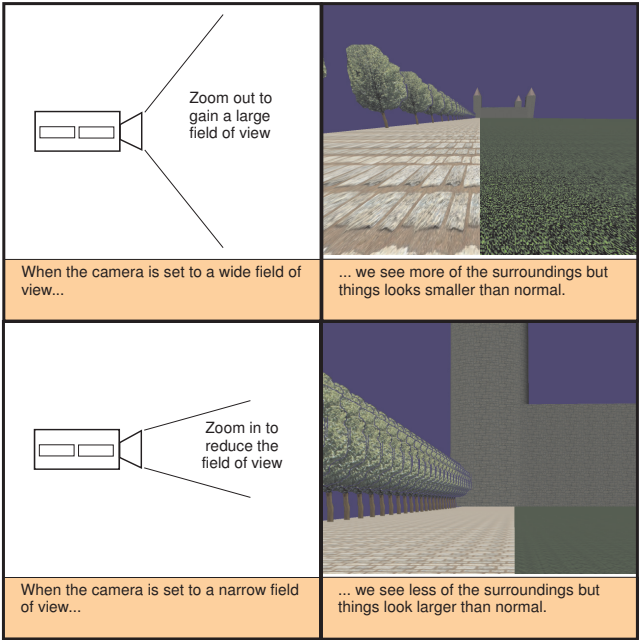
In your last program, correct the distortion shown in the camera display.

The SET CAMERA FOV Statement

Real cameras often come with a zoom lens. This allows the user to zoom in on distant objects or zoom out to get a wide-angle shot (useful when trying to record in small spaces). Just exactly how much can be included in a shot is known as the **field of view** (or FOV) and is measured as the angle between what appears on the left edge of the screen and the right (see FIG-33.22). When zooming in on distant objects, we have a very limited field of view; in wide-angle mode, we have a wide field of view.

FIG-33.22

The Effect of Changing
the Field of View



Our own virtual camera can perform the same trick using the SET CAMERA FOV statement which has the format shown in FIG-33.23.

FIG-33.23

The SET CAMERA
FOV Statement



In the diagram:

camno

is an integer value specifying the camera whose

field of view is to be defined. If this option is omitted the currently active camera is assumed.

angle

is a real number specifying the angle of field of view.

The program in LISTING-33.1 demonstrates the effect of changes to the field of view. It starts with an angle of 180° and reduces to 5° effectively zooming in on one of the turrets on the castle. For comparison, a second loop in the program actually moves the camera towards the turret without changing the field of view from a starting value of 55° .

LISTING-33.1

Using Zoom

```
#INCLUDE "Castle.dba"
REM *** Set screen resolution ***
SET DISPLAY MODE 1280,1024,32
REM *** Draw the castle ***
DrawCastle()
REM *** Position camera ***
POSITION CAMERA 0,8,0
REM *** Set the camera screen area ***
SET CAMERA VIEW 0,0,1279,639

REM *** Set the aspect ratio ***
SET CAMERA ASPECT 2/1

REM *** Point camera at turret ***
POINT CAMERA -10,75,200

REM *** Zoom in on turret ***
WAIT KEY
FOR angle = 180 TO 5 STEP -1
    SET CAMERA FOV angle
    WAIT 50
NEXT angle

REM *** Move camera towards turret ***
WAIT KEY
SET CAMERA FOV 55
POINT CAMERA -10,75,200
FOR move = 1 TO 580
    MOVE CAMERA 1
    WAIT 20
NEXT move

REM *** End program ***
WAIT KEY
END
```

Activity 33.10

Type in and test the program given in LISTING-33.1 (*camera02.dbpro*).

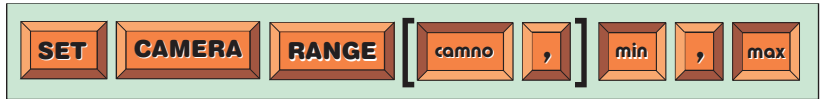
The SET CAMERA RANGE Statement

It is possible to specify the range of the camera's vision. This means that objects which are too close to the camera, or too far away, become invisible and hence do not appear on the screen.

This effect is achieved using the SET CAMERA RANGE statement which has the format shown in FIG-33.24.

FIG-33.24

The SET CAMERA
RANGE Statement



In the diagram:

camno

is an integer value specifying the camera affected.
If no camera number is given, the active camera
is assumed.

min

is a real number specifying the closest distance
at which an object will be visible to the camera.

max

is a real number specifying the furthest distance
at which an object will be visible to the camera.

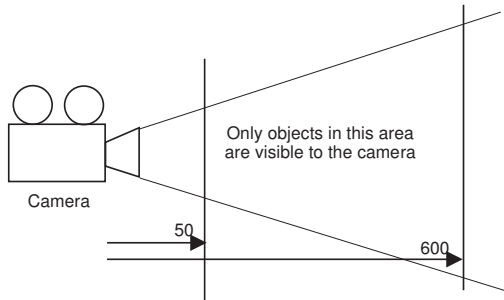
For example, the statement

```
SET CAMERA RANGE 50, 600
```

would create the effect shown in FIG-33.25.

FIG-33.25

Restricting the Visible
Area to Between 50 and
600 Units from the
Camera



Activity 33.11

In your last program, add the line

```
SET CAMERA RANGE 50, 600
```

immediately before the first FOR loop and observe how this affects the view
displayed on the screen.

Summary

- Whenever 3D objects are used in a DarkBASIC Pro program, a default camera is automatically created.
- The image shown on the screen is the picture transmitted by the default camera.
- The camera is not a visible object.
- Use POSITION CAMERA to move the camera to a specific point.
- Use MOVE CAMERA to move the camera by a specified distance. Movement can be forwards or backwards.

- Use POINT CAMERA to aim the camera at a specific position.
- Use ROTATE CAMERA to rotate the camera to absolute angles about all of its three axes.
- Use SET CAMERA ROTATION to specify the order of axes rotations.
- Use XROTATE CAMERA to rotate the camera to an absolute angle about the x-axis only.
- Use YROTATE CAMERA to rotate the camera to an absolute angle about the y-axis only.
- Use ZROTATE CAMERA to rotate the camera to an absolute angle about the z-axis only.
- Use PITCH CAMERA to rotate the camera by a number of degrees about its x-axis.
- Use TURN CAMERA to rotate the camera by a number of degrees about its y-axis.
- Use ROLL CAMERA to rotate the camera by a number of degrees about its z-axis.
- Use CAMERA POSITION to retrieve the camera's current position.
- Use CAMERA ANGLE to retrieve the camera's angle of rotation about a specific axis.
- Use SET CAMERA VIEW to set how much of the screen the camera's picture occupies.
- Use SET CAMERA ASPECT to set the aspect ratio of the camera's picture.
- Use SET CAMERA FOV to set the camera's field of view. This is equivalent to using a zoom lens on a standard camera.
- Use SET CAMERA RANGE to set the nearest and furthest point visible to the camera.

Controlling Camera Movement

Introduction

Although we've already covered many commands that allow us to move, point and rotate the camera, we still need to develop a few techniques to cover special situations.

Automatic Camera Placement

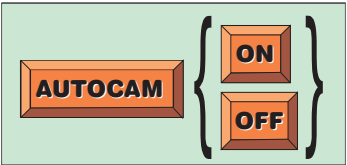
The AUTOCAM Statement

The initial position of the camera is determined by the size and position of the 3D objects that appear in the program. If we create a large cube, the camera will be placed further back (its z-ordinate becoming more negative); with a smaller cube the camera would be placed closer.

If, for some reason, we do not want automatic positioning to be implemented, we can force the camera to start at position (0,0,0) using the line AUTOCAM OFF. To return to automatic positioning use AUTOCAM ON. The AUTOCAM statement has the format shown in FIG-33.26.

FIG-33.26

The AUTOCAM OFF
Statement



The program in LISTING-33.2 demonstrates the effect of automatic camera positioning by displaying the camera's position for a cube whose size can be set.

LISTING-33.2

Automatic Camera
Positioning

```
REM *** Set screen resolution ***
SET DISPLAY MODE 1280,1024,32
REM *** Get size of cube ***
INPUT "Enter size of cube (2 to 100):",size
REM *** Create cube ***
MAKE OBJECT CUBE 1, size
REM *** Display position of camera ***
DO
    SET CURSOR 100,100
    PRINT "(" , CAMERA POSITION X() , "," , CAMERA POSITION Y() ,
    ↵ ", " , CAMERA POSITION Z() , ")" "
LOOP
REM *** End program ***
END
```

Activity 33.12

Type in and test the program in LISTING-33.2 (*camera03.dbpro*).

Modify the program by inserting the line

AUTOCAM OFF

immediately after the SET DISPLAY MODE statement.

How does this affect the camera's position?

Following the Action

In the next program (see LISTING-33.3) we see a strange object crash to the ground outside the castle. In this first version of the program the camera is stationary and we see only part of the object's journey.

LISTING-33.3

Creating a Falling Object

```
#INCLUDE "Castle.dba"
#CONSTANT artifact 200
#CONSTANT arttexture 200

REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP RGB(64,0,128)
BACKDROP ON

REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 0,8,10

REM *** Draw castle ***
DrawCastle()

REM *** Show falling object ***
FallingObject()

REM *** End program ***
WAIT KEY
END

FUNCTION FallingObject()
    REM *** Create textured transparent sphere ***
    MAKE OBJECT SPHERE artifact,10
    LOAD IMAGE "lattice.bmp",arttexture
    TEXTURE OBJECT artifact,arttexture
    SET OBJECT TRANSPARENCY artifact, 1
    REM *** Position artifact out in sky ***
    POSITION OBJECT artifact, -200,300,900
    REM *** Point the object towards the ground ***
    POINT OBJECT artifact,300,0,30
    REM *** Move object, stopping when it hits the ground ***
    DO
        MOVE OBJECT artifact,8
        IF OBJECT POSITION Y(artifact) <= 2
            EXIT
        ENDIF
    LOOP
ENDFUNCTION
```

You may want to put a WAIT 20 statement at the end of the DO..LOOP if your machine is a fast one.

Activity 33.13

Type in and run the program given above (*camera04.dbpro*).

We can modify the program so that the camera is always pointing at the falling object by adding the line

```
POINT CAMERA OBJECT POSITION X(artifact),
    ↵ OBJECT POSITION Y(artifact), OBJECT POSITION Z(artifact)
```

after the line `MOVE OBJECT artifact,8` in the *FallingObject()* function.

Activity 33.14

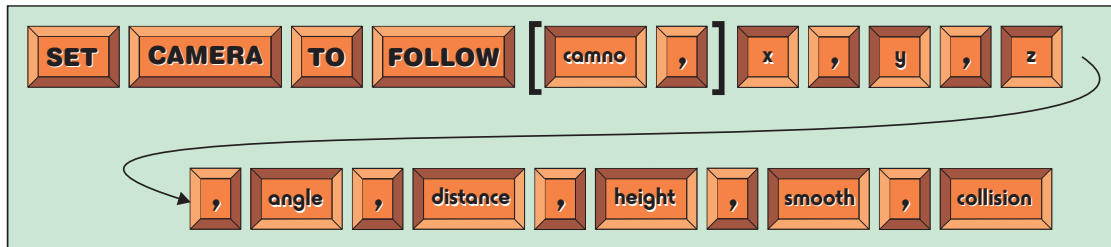
Modify your previous program as described above and observe how this affects the behaviour of the camera.

The SET CAMERA TO FOLLOW Statement

FIG-33.27

The SET CAMERA TO FOLLOW Statement

An even more dramatic effect can be produced by getting the camera to move along with an object, keeping that object in the centre of the camera output. This can be achieved using the SET CAMERA TO FOLLOW statement which has the format shown in FIG-33.27.



In the diagram:

camno

is an integer value specifying the camera affected. If no camera number is given, the active camera is assumed.

x,y,z

is a set of real numbers giving the coordinates of the position to which the camera is to point.

angle

is a real number giving the angle (measured from the z-axis) the camera is to maintain in relation to the the point (x,y,z).

distance

is a real number giving the distance the camera is to be from the point (x,y,z).

height

is a real number giving the camera's height above (or below) the point (x,y,z).

smooth

is a real number specifying how smoothly the camera will move from its current position to the new position it requires to be in to meet the specifications given by the SET CAMERA TO FOLLOW statement.

collision

is 0 or 1 and is used to specify how the camera is to react to colliding with a static collision box.

Collision boxes are discussed in a later chapter.

This is one of the most complex statements in DarkBASIC Pro, so it's worth spending a bit of time finding out just how each value in the statement affects the results you achieve. To demonstrate how it works we'll track the falling object of our previous program.

The x,y,z values give the position to be tracked, so in this case that means the

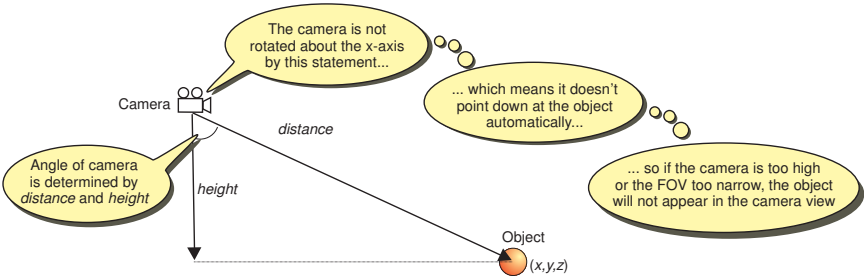
coordinates of the falling object which we can get using:

```
OBJECT POSITION X(artifact), OBJECT POSITION Y(artifact),  
↓  
OBJECT POSITION Z(artifact)
```

The *distance* and *height* values specify how far the camera is from the point being tracked and the height above that point (see FIG-33.28).

FIG-33.28

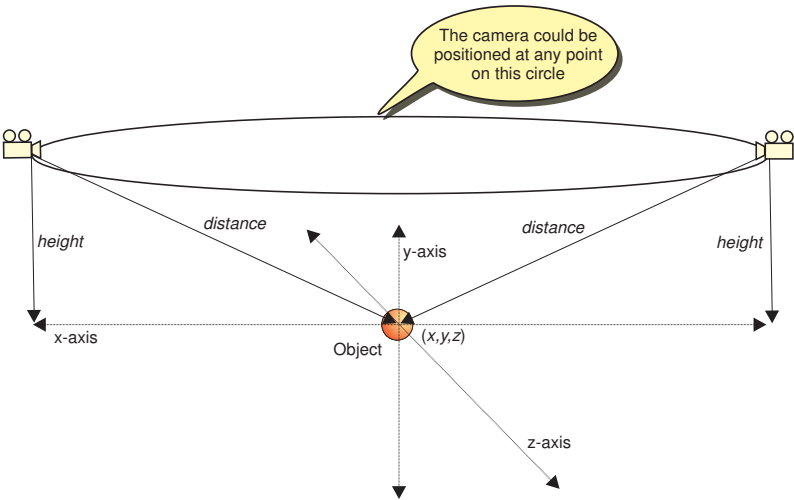
The Camera Distance and Height



This is not enough information to specify a single position for the camera, since any point on a circle which is *distance* units from the object and *height* units above the object will meet the criteria specified (see FIG-33.29).

FIG-33.29

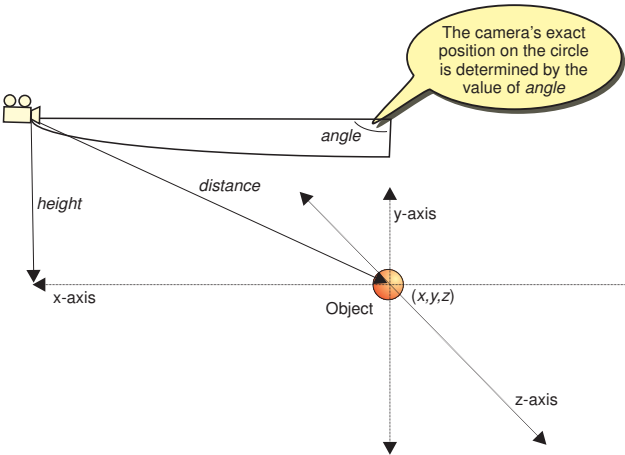
Possible Camera Positions



The final value required to create a unique position for the camera is the *angle* value which specifies how far round the circle the camera is to be placed (see FIG-33.30).

FIG-33.30

The Exact Position is Determined by *angle*



Before we look at the purpose of the last two values used in the SET CAMERA TO FOLLOW statement, we'll get a feel for how the statement works by modifying the last program so that the *FallingObject()* function is now coded as:

```
FUNCTION FallingObject()  
    REM *** Create textured transparent sphere ***  
    MAKE OBJECT SPHERE artifact,10  
    LOAD IMAGE "lattice.bmp",arttexture  
    TEXTURE OBJECT artifact,arttexture  
    SET OBJECT TRANSPARENCY artifact, 1  
    REM *** Position artifact out in sky ***  
    POSITION OBJECT artifact, -200,300,900  
    REM *** Point the object towards the ground ***  
    POINT OBJECT artifact,300,0,30  
    REM *** Move object, stopping when it hits the ground ***  
    DO  
        MOVE OBJECT artifact, 8  
        SET CAMERA TO FOLLOW OBJECT POSITION X(artifact),  
        ↵ OBJECT POSITION Y(artifact),OBJECT POSITION Z(artifact),  
        ↵ 0,20,0,1.0,1  
        IF OBJECT POSITION Y(artifact) <= 2  
            EXIT  
        ENDIF  
    LOOP  
ENDFUNCTION
```

Notice that the angle and height values are set to zero, while the distance setting is 20.

Activity 33.15

Modify program *camera04.dbpro* so that its *FallingObject()* function matches the code given above.

Modify the program as follows, running the program after each change:

Set *distance* to 100
Set *height* to 20
Set *angle* to 90

The *smooth* parameter determines how smoothly the camera moves from its current position to the position required by the latest call to SET CAMERA TO FOLLOW statement. Using a value of 1.0 the camera will jump almost instantly to the required position, no matter what its current location. On the other hand, by using a value of 100.0, the camera may take hundreds of frames (that is, several seconds) to move into position.

Activity 33.16

Set the *smooth* value in your last program's SET CAMERA TO FOLLOW statement to 100.0 and re-run the program.

What difference does this make to the camera's movements?

Change the *smooth* value to 20 and run the program again.

In our example, the smoothing factor affected the camera's movement from its original position at (0,8,10) to its new position behind the sphere. The effect of smoothing is also apparent if the object being tracked moves erratically in

unpredictable directions and varying speeds.

We can even use this instruction with a high smoothing factor to make the camera move slowly along a path to a given point. For example, the lines

```
POSITION CAMERA 0,8,10
DO
    SET CAMERA TO FOLLOW -20,200,900,1800,10,10,100.0,1
LOOP
```

will move the camera smoothly from the point (0,8,10) to within 10 units of the point (-20,200,900).

Activity 33.17

Change the main section of your previous program to read

```
#INCLUDE "Castle.dba"
#CONSTANT artifact 200
#CONSTANT arttexture 200

SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP RGB(64,0,128)
BACKDROP ON
AUTOCAM OFF
DrawCastle()
POSITION CAMERA 0,8,10
DO
    SET CAMERA TO FOLLOW -20,200,900,180,10,10,100.0,1
LOOP
END
```

and check out how the camera moves and rotates between its starting and finishing points.

You may have noticed that the camera moves straight through the first turret! Unless it is playing the part of a ghostly apparition, we probably don't want this to happen. To prevent this we use the final parameter of the SET CAMERA TO FOLLOW statement which allows the camera to react to collisions. However, we need to know how to handle collisions in general before examining this option, so we'll leave it for a later chapter.

Giving the Player Control of the Camera

The CONTROL CAMERA USING ARROWKEYS Statement

The simplest way to give the player control of the camera position is to use the CONTROL CAMERA USING ARROWKEYS statement. This allows the player to rotate the camera about the y-axis using the left and right arrow keys and to move forwards and backwards using the up and down arrow keys. The statement also allows the speed of the camera's response to these keys to be specified. This statement has the format shown in FIG-33.31

FIG-33.31

The CONTROL CAMERA USING
ARROWKEYS Statement



In the diagram:

<i>camno</i>	is an integer value specifying the camera to be used.
<i>move</i>	is a real number specifying the speed at which the camera moves in response to pressing the up and down arrow keys.
<i>turn</i>	is a real number specifying the speed at which the camera rotates in response to pressing the left and right arrow keys.

For example, we could control the movement of camera 0, allowing a move of 0.5 units each time an up or down arrow key is pressed and with a rotation of 0.1° every time a left or right arrow key is pressed using the line:

```
CONTROL CAMERA USING ARROW KEYS 0,0.5,0.1
```

The statement must be placed in a loop structure (usually a `DO..LOOP`) so that the statement is continually executed.

In LISTING-33.4 the user is allowed to move about the castle area by controlling the camera.

LISTING-33.4

Basic User Camera
Control

```
#INCLUDE "Castle.dba"
#CONSTANT artifact 200
#CONSTANT arttexture 200

REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP RGB(64,0,128)
BACKDROP ON

REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 0,8,10

REM *** Draw the castle and show falling object ***
DrawCastle()
FallingObject()

REM *** Re-position the camera ***
POSITION CAMERA 0,8,10
POINT CAMERA 0,8,200

REM *** Give user camera control ***
DO
    CONTROL CAMERA USING ARROWKEYS 0,1,0.5
LOOP

REM *** End program ***
END
```

LISTING-33.4
(continued)

Basic User Camera
Control

```
FUNCTION FallingObject()  
  REM *** Create textured transparent sphere ***  
  MAKE OBJECT SPHERE artifact,10  
  LOAD IMAGE "lattice.bmp",arttexture  
  TEXTURE OBJECT artifact,arttexture  
  SET OBJECT TRANSPARENCY artifact, 1  
  
  REM *** Position artifact out in sky ***  
  POSITION OBJECT artifact, -200,300,900  
  
  REM *** Point the object towards the ground ***  
  POINT OBJECT artifact,300,0,30  
  
  REM *** Move object, stopping when it hits the ground ***  
  DO  
    MOVE OBJECT artifact, 8  
    SET CAMERA TO FOLLOW OBJECT POSITION X(artifact),  
    ↵ OBJECT POSITION Y(artifact),OBJECT POSITION Z(artifact),  
    ↵ 90,100,20,20,1  
    IF OBJECT POSITION Y(artifact) <= 2  
      EXIT  
    ENDIF  
  LOOP  
ENDFUNCTION
```

Activity 33.18

Type in and test the program in LISTING-33.4 (*camera05.dbpro*).

Can you move inside the castle?

As you can see from the results of the Activity, the player takes on ghost-like qualities; being able to walk through solid objects! Luckily, there's an easy way to stop this.

The AUTOMATIC CAMERA COLLISION Statement

By using the AUTOMATIC CAMERA COLLISION statement we can stop a camera moving through a 3D object. This is particularly useful when the user is controlling camera movement. The statement has the format shown in FIG-33.32.

FIG-33.32

The AUTOMATIC CAMERA
COLLISION Statement



In the diagram:

- camno* is an integer value specifying the camera involved.
- radius* is a real number giving a collision radius. If the camera comes within this distance of any 3D object a collision between that object and the camera will be assumed.
- response* is 0 or 1. Using the value zero, the camera reacts to a collision with another object by "sliding off

to the side" of the object - this is known as a sliding collision. When set to 1, the camera halts when a collision is detected.

For example, we could make camera zero halt when it comes within 2 units of an object using the line:

```
AUTOMATIC CAMERA COLLISION 0,2,1
```

Activity 33.19

By placing the line

```
AUTOMATIC CAMERA COLLISION 0,1,0
```

before the DO loop in the main section of your last program, make the user-control camera slide past any collisions.

Controlling the Camera with the Mouse

Arrow key camera control is of limited use since the camera can only be rotated about the y-axis. Also, camera movement is only in the x-z plane (or a plane parallel to this). To gain greater control, we'll have to write our own code.

We'll start by allowing the user to rotate the camera about both the x and y axes using mouse movement. This is done by a function, *PointCameraUsingMouse()*, which rotates a specified camera in response to mouse movement. The first version of the routine is given below:

```
FUNCTION PointCameraUsingMouse(camno)
  REM *** Get current camera angles ***
  XcurrentAngle# = CAMERA ANGLE X(camno)
  YcurrentAngle# = CAMERA ANGLE Y(camno)
  REM *** Calculate the new angle based on mouse movement ***
  XnewAngle# = XcurrentAngle# + MOUSEMOVEY()
  YnewAngle# = YcurrentAngle# + MOUSEMOVEX()
  REM *** Rotate camera ***
  XROTATE CAMERA camno,XnewAngle#
  YROTATE CAMERA camno,YnewAngle#
ENDFUNCTION
```

The routine stores the current angles of rotation and then calculates new angles based on the movement of the mouse. Notice that mouse movement in the y direction changes the angle of rotation about the x-axis, while mouse movement in the x direction modifies the angle about the y-axis!

The routine is put to use in LISTING-33.5 which allows the user total freedom of rotation in both the x and y axes.

LISTING-33.5

Camera Control Using
the Mouse

```
#INCLUDE "Castle.dba"
REM *** set up screen ***
SET DISPLAY MODE 1280,1024,32
AUTOCAM OFF
DrawCastle()
REM *** Re-position camera ***
POSITION CAMERA 0,8,10
POINT CAMERA 0,8,200
```

continued on next page

LISTING-33.5

(continued)

Camera Control Using
the Mouse

```
REM *** Give user camera control ***
DO
    PointCameraUsingMouse(0)
LOOP

REM *** End program ***
END

REM *** Rotates specified camera ***
FUNCTION PointCameraUsingMouse(camno)
    REM *** Get current camera angles ***
    XcurrentAngle# = CAMERA ANGLE X(camno)
    YcurrentAngle# = CAMERA ANGLE Y(camno)
    XnewAngle# = XcurrentAngle# + MOUSEMOVEY()
    YnewAngle# = YcurrentAngle# + MOUSEMOVEX()
    REM *** Rotate camera ***
    XROTATE CAMERA camno, XnewAngle#
    YROTATE CAMERA camno, YnewAngle#
ENDFUNCTION
```

Activity 33.20

Type in and test the program given in LISTING-33.5 (*camera06.dbpro*).

Are there any problems with the rotation control?

Rotation is probably a bit too sensitive to mouse movement but we can overcome this by reducing the effect of the mouse move values by changing the new angle calculation statements to:

```
XnewAngle# = XcurrentAngle# + MOUSEMOVEY() * 0.1
YnewAngle# = YcurrentAngle# + MOUSEMOVEX() * 0.1
```

Rotation about the y-axis should be kept in the range 0 to 359.9 and we can do this by applying the WRAPVALUE statement before assigning the value of the new angle:

```
YnewAngle# = WRAPVALUE(YcurrentAngle# + MOUSEMOVEX() * 0.1)
```

Given the restriction of a human's head movements, rotation about the x-axis should probably remain within the range -90 to +90 so this angle should be calculated using the lines:

```
XnewAngle# = XcurrentAngle# + MOUSEMOVEY() * 0.1
IF XnewAngle# < -90
    XnewAngle# = -90
ELSE
    IF XnewAngle# > 90
        XnewAngle# = 90
    ENDIF
ENDIF
ENDIF
```

Activity 33.21

Make the changes described above to your previous program and check out how this affects the performance of the camera.

Now that we can point the camera in any direction, the next thing we need to do is

move the camera. We already know how to do this using the CONTROL CAMERA USING THE ARROW KEYS statement.

Activity 33.22

Add the line

```
CONTROL CAMERA USING ARROW KEYS 0,0.5,0
```

to the DO..LOOP structure in your last program.

Does this change create any problems?

As we can see from the last Activity, the CONTROL CAMERA USING ARROWKEYS statement always returns the camera to zero degrees rotation about the x-axis, effectively stopping us from pointing the camera up or down.

One way to deal with this is to write our own movement routine. The routine given below takes the camera involved and the distance moved each time as parameters. The camera moves in the direction it is pointing by pressing the left mouse button; if the right mouse button is pressed, the camera moves backwards.

```
FUNCTION MoveCameraUsingMouse(camno, distance#)
  IF MOUSECLICK() = 1
    MOVE CAMERA camno,distance#
  ELSE
    IF MOUSECLICK() = 2
      MOVE CAMERA camno,-distance#
    ENDIF
  ENDIF
ENDFUNCTION
```

Activity 33.23

Add the above function to your last program and replace the CONTROL CAMERA USING ARROWKEYS line with the code

```
MoveCameraUsingMouse(0,1)
```

This gives us more freedom of movement than we might want for a player, but it is useful when checking the layout of our virtual world.

Activity 33.24

By temporarily rotating the camera to zero degrees about the x-axis during movement, modify the *MoveCameraUsingMouse()* function so that movement is always parallel to the x-z plane irrespective of camera elevation.

To stop the camera moving through solid objects, add an AUTOMATIC CAMERA COLLISION statement immediately before the DO..LOOP structure in the main section of the program.

To stop the camera moving outside the modelled area, modify the *MoveCameraUsingMouse()* function to limit the position of the camera to be within the area bounded by the x-z points (-50,0) (500,700).

Activity 33.25

These mouse-camera control functions will prove useful elsewhere, so it will be worthwhile saving these functions from the last program in a separate file then using the `#INCLUDE` statement to add them to future programs.

Remove the code which limits movement to the area `(-50,0),(500,700)` in *MoveMouseUsing Camera()*.

Create a new project called *Camera.dbpro*.

Copy the code for the two functions, *MoveCameraUsingMouse()* and *PointCameraUsingMouse()*, into the program file and then save and close the project.

Summary

- Use `AUTOCAM OFF` to stop the camera being positioned automatically by the program.
- Use `AUTOCAM ON` to reinstate automatic camera positioning.
- Use `SET CAMERA TO FOLLOW` to make the camera move smoothly so that it is positioned to focus on a specific spot.
- Use `CONTROL CAMERA USING ARROWKEYS` to allow the user to control x-z movement of the camera using the arrow keys.
- Use `AUTOMATIC CAMERA COLLISION` to stop the camera moving through solid objects.
- For more flexible camera control, write your own functions.

Multiple Cameras

Introduction

So far we have used only the default camera (camera zero) which exists in every DarkBASIC Pro 3D program, but it is possible to use multiple cameras. In the statements that follow we'll see how this is done and how we can use several cameras to enhance a game.

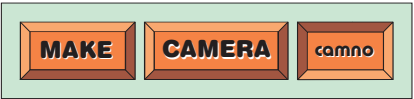
Using Additional Cameras

The MAKE CAMERA Statement

We can create a new camera using the MAKE CAMERA statement which has the format shown in FIG-33.33.

FIG-33.33

The MAKE CAMERA Statement



In the diagram:

camno

is an integer value specifying the ID to be assigned to the new camera. No two cameras may be assigned the same ID.

For example, we could create a camera with ID 1 using the statement:

```
MAKE CAMERA 1
```

When a new camera is created, it is positioned at (0,0,0) and points in a positive direction along the z-axis. The output from the camera is automatically displayed on the screen, replacing the display from any previous camera.

The program in LISTING-33.6 demonstrates this effect.

LISTING-33.6

Creating a new Camera

```
#INCLUDE "Castle.dba"

REM *** set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP RGB(64,0,128)
BACKDROP ON
AUTOCAM OFF

DrawCastle()

REM *** Re-position the default camera ***
POSITION CAMERA -20,8,10
POINT CAMERA 0,8,200

REM *** Make new camera ***
WAIT KEY
MAKE CAMERA 1

REM *** End program ***
WAIT KEY
END
```

Activity 33.26

Type in and test the program in LISTING-33.6 (*camera07.dbpro*).

Why does the grass and path not show in the image from the second camera?

Notice that the background has turned green! Camera zero defaults to a blue background, while camera 1's default background is always green.

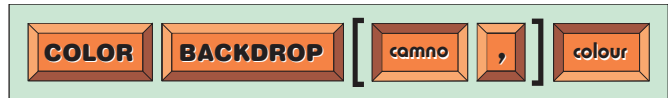
Why did the background change colour when we saw the output from the second camera? It turns out that the **COLOR BACKDROP** statement specifies the background colour for the current camera only. In fact, the **BACKDROP** and **COLOR BACKDROP** statements have a syntax slightly different from that given back in volume 1, because both allow a camera to be specified. The updated version of each statement is given below.

The COLOR BACKDROP Statement

This statement can be used to specify the background colour displayed by a specific camera. The statement has the format shown in FIG-33.34.

FIG-33.34

The COLOR
BACKDROP Statement



In the diagram:

camno

is an integer value giving the ID of the camera whose background colour is to be changed. If this value is omitted, it is the current camera that is affected by the statement.

colour

is an integer value representing the new background colour.

To set the background colour for camera 1 to black we would use the statement

```
COLOR BACKGROUND 1,0
```

but for other colours we would probably use an RGB expression, as in the line:

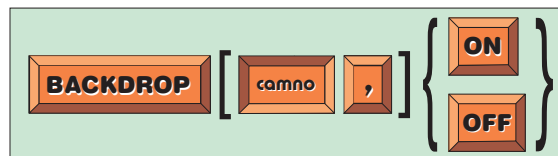
```
COLOR BACKGROUND 1,RGB (255,255,0)
```

The BACKDROP Statement

The background colour specified using the **COLOR BACKDROP** statement is activated (or deactivated) using the **BACKDROP** statement which has the format shown in FIG-33.35.

FIG-33.35

The BACKDROP
Statement



In the diagram:

camno

is an integer value giving the ID of the camera whose background colour is to be (de)activated. If this value is omitted, it is the current camera that is affected by the statement.

ON

Use this option to activate the background colour.

OFF

Use this option to deactivate the background colour and return to the default colour.

Activity 33.27

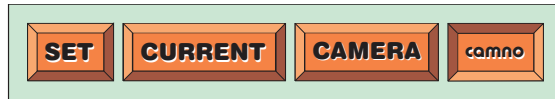
Modify your last program so that the background colour for camera 1 matches that of camera zero.

The SET CURRENT CAMERA Statement

When we have more than one camera available, we can choose which one is the active camera using the SET CURRENT CAMERA statement which has the format shown in FIG-33.36.

FIG-33.36

The SET CURRENT CAMERA Statement



In the diagram:

camno

is an integer value specifying which camera is to be the active camera.

When a program begins, camera zero is the active camera. Camera zero remains the active camera even after other cameras have been created. Once a new active camera has been specified using SET CURRENT CAMERA, any subsequent camera-related statement which does not explicitly specify a camera number will be assumed to be acting on this camera.

In LISTING-33.7 the previous program is modified so that camera 1 is moved above and to the back of the castle. The view from camera 1 is activated by a key press.

LISTING-33.7

Setting the Current Camera

```
#INCLUDE "Castle.dba"

REM *** set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP RGB(64,0,128)
BACKDROP ON
AUTOCAM OFF

DrawCastle()

REM *** Re-position camera zero ***
POSITION CAMERA 0,8,10
POINT CAMERA 0,8,200

REM *** Make and position new camera ***
WAIT KEY
```

continued on next page

LISTING-33.7

(continued)

Setting the Current
Camera

```
MAKE CAMERA 1
SET CURRENT CAMERA 1
POSITION CAMERA 50,250,900
POINT CAMERA 200,0,0

REM *** End program ***
WAIT KEY
END
```

Activity 33.28

Modify your previous program to match LISTING-33.7.

The DELETE CAMERA Statement

A camera's details can be deleted from RAM using the DELETE CAMERA statement which has the format shown in FIG-33.37.

FIG-33.37

The DELETE CAMERA
Statement



In the diagram:

camno

is an integer value specifying the camera to be deleted. Camera zero cannot be deleted.

For example, we could delete camera 1 using the statement:

```
DELETE CAMERA 1
```

Switching Between Cameras

The current version of DarkBASIC Pro (version 1.060) has no direct way of switching between cameras. That is to say, if the computer screen is currently showing the output from camera 2, there is no simple way to switch back to showing the output from camera 1.

Since the output shown does change when a new camera is created, the only way to switch between cameras is to delete and then recreate a camera. We also need to save the position of the camera and the direction in which it is pointing so that the recreated camera is correctly placed.

A function to perform this task is given below:

```
FUNCTION SwitchCamera(camno)
  REM *** Save camera position ***
  x# = CAMERA POSITION X(camno)
  y# = CAMERA POSITION Y(camno)
  z# = CAMERA POSITION Z(camno)
  REM *** Save camera direction ***
  angx# = CAMERA ANGLE X(camno)
  angy# = CAMERA ANGLE Y(camno)
  angz# = CAMERA ANGLE Z(camno)
  REM *** Delete and recreate camera ***
  DELETE CAMERA camno
  MAKE CAMERA camno
  REM *** Set camera as active camera ***
  SET CURRENT CAMERA camno
  REM *** Restore original position and direction ***
```

```

        POSITION CAMERA camno,x#,y#,z#
        ROTATE CAMERA camno,angx#,angy#,angz#
    ENDFUNCTION

```

Obviously, the camera in question must already exist before the function is called. Ideally, the function should start with the lines

```

        IF NOT CAMERA EXIST(camno)
            RETURN
        ENDIF

```

but there is currently no CAMERA EXIST statement in DarkBASIC Pro.

After the *SwitchCamera()* function has been called, it will be necessary to reset the background colour. For example, to switch to camera 2 with a red background we would use the lines:

```

    SwitchCamera(2)
    COLOR BACKDROP RGB(64,0,128)
    BACKDROP ON

```

The program in LISTING-33.8 demonstrates the use of camera switching by allowing the user to switch between cameras 1 and 2 using the "1" and "2" keys.

LISTING-33.8

Switching Cameras

```

#include "Castle.dba"

REM *** set up screen and use camera 1***
SET DISPLAY MODE 1280,1024,32
MAKE CAMERA 1
SET CURRENT CAMERA 1
COLOR BACKDROP RGB(64,0,128)
BACKDROP ON
AUTOCAM OFF

DrawCastle()

REM *** Re-position camera 1 ***
POSITION CAMERA 0,8,10
POINT CAMERA 0,8,200

REM *** Make and position camera 2 ***
WAIT KEY
MAKE CAMERA 2
SET CURRENT CAMERA 2
COLOR BACKDROP RGB(64,0,128)
BACKDROP ON
POSITION CAMERA 50,250,900
POINT CAMERA 200,0,0

REM *** Allow user to switch between cameras ***
DO
    REPEAT
        c$ = INKEY$()
        UNTIL c$ = "1" OR c$ = "2"
        SwitchCamera(VAL(c$))
        COLOR BACKDROP RGB(64,0,128)
        BACKDROP ON
    LOOP

REM *** End program ***
END

```

continued on next page

LISTING-33.8

(continued)

SwitchingCameras

```
FUNCTION SwitchCamera(camno)
    REM *** Save camera details ***
    x# = CAMERA POSITION X(camno)
    y# = CAMERA POSITION Y(camno)
    z# = CAMERA POSITION Z(camno)

    REM *** Save camera direction ***
    angx# = CAMERA ANGLE X(camno)
    angy# = CAMERA ANGLE Y(camno)
    angz# = CAMERA ANGLE Z(camno)

    REM *** Delete and recreate camera ***
    DELETE CAMERA camno
    MAKE CAMERA camno

    REM *** Set camera as active camera ***
    SET CURRENT CAMERA camno

    REM *** Restore original position and direction ***
    POSITION CAMERA camno,x#,y#,z#
    ROTATE CAMERA camno,angx#,angy#,angz#
ENDFUNCTION
```

Activity 33.29

Type in and test the program in LISTING-33.8 (*camera08.dbpro*).

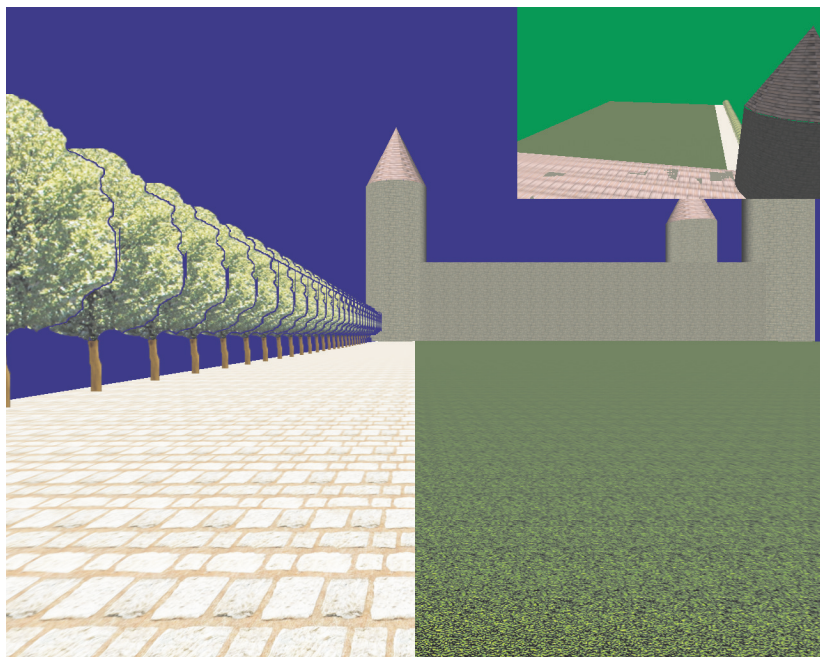
The *SwitchCamera()* function will be useful elsewhere. Add this function to the *Camera.dbpro* project you created back in Activity 33.25.

Multiple Camera Output

It is possible to display the output from two or more cameras at the same time by defining different output areas for each camera using the SET CAMERA VIEW statement. The output shown in FIG-33.38 is produced by the program given in LISTING-33.9.

FIG-33.38

Simultaneous Output
from Multiple Cameras



LISTING-33.9

Picture-in-Picture

```
#INCLUDE "Castle.dba"
REM *** set up screen ***
SET DISPLAY MODE 1280,1024,32
AUTOCAM OFF

DrawCastle()

REM *** Re-position camera ***
POSITION CAMERA 0,8,10
POINT CAMERA 0,8,200

REM *** Create a second camera ...***
MAKE CAMERA 1
POSITION CAMERA 1,50,250,900
POINT CAMERA 1,200,0,0

REM *** ... with its own output area ***
SET CAMERA VIEW 1, 800,0,1279,300

REM *** End program ***
WAIT KEY
END
```

Activity 33.30

Type in and test the program in LISTING-33.9 (*camera09.dbpro*).

Modify the program so that the user can control movement of camera 0 using the mouse.

Modify the program again so that camera 1 slowly rotates to the left.

The CLEAR CAMERA VIEW Statement

We can clear the screen area to which a camera displays its output using the CLEAR CAMERA VIEW statement which has the format shown in FIG-33.39.

FIG-33.39

The CLEAR CAMERA VIEW Statement



In the diagram:

camno

is an integer value specifying the camera whose view port is to be cleared. If no camera ID is given, the active camera is assumed.

colour

is an integer value specifying the colour to be used when clearing the view port.

We could clear the view area of camera 1 to red using the line:

```
CLEAR CAMERA VIEW 1, RGB(255,0,0)
```

Since the camera's output is recalculated for every frame, and the effect of the CLEAR CAMERA VIEW statement only lasts for a single frame, there doesn't seem to be much use for this statement.

Summary

- Use `MAKE CAMERA` to create a new camera.
- When a new camera is created, the output of that camera is displayed on the screen.
- A new camera does not automatically become the active camera.
- Use `COLOR BACKDROP` and `BACKDROP ON` to select the background colour for a camera's output.
- Use `SET CURRENT CAMERA` to set the active camera.
- Any statement in which the camera number is omitted is assumed to relate to the active camera.
- Use `DELETE CAMERA` to delete a camera from RAM.
- Camera zero cannot be deleted.
- The output of multiple cameras can appear on the screen at the same time by allocating different areas of the screen to each camera.
- Use `CLEAR CAMERA VIEW` to clear the camera's output area using a specified colour.

Advanced Camera Techniques

Introduction

Some camera-related statements can be used in conjunction with other statements to recreate effects such as mirrors or televisions or to show our character carrying a weapon.

The Statements

The SET CAMERA TO IMAGE Statement

A rather neat trick is to display the output from a camera on the surface of an object. This allows us to create things like mirrors or TV monitors. To achieve this affect we need to first transfer the camera's output to an image object and then texture the 3D object with that image. The first stage in this process is achieved using the SET CAMERA TO IMAGE statement whose format is given in FIG-33.40.

FIG-33.40 The SET CAMERA TO IMAGE Statement



In the diagram:

<i>camno</i>	is an integer value specifying the camera whose output is to be mapped onto an image object.
<i>imgno</i>	is an integer value specifying the ID of the image to which camera output is to be sent. If an image with this ID number does not already exist, it will be created automatically.
<i>width</i>	is an integer value giving the width of the image object in pixels. This value must be a power of 2 (that is, 64, 128, 256, etc.)
<i>height</i>	is an integer value giving the height of the image object in pixels. This value must be a power of 2.

For example, we could output camera 1's display to image 10 which is set to a width of 128 by 128 using the line:

```
SET CAMERA TO IMAGE 1,10,128,128
```

If image 10 does not exist when the statement is executed, it will be created. However, if it does exist, its contents will be modified by this statement.

Next we need to use the image to texture a 3D object. For example, we could texture object 5 with our new image using the line:

```
TEXTURE OBJECT 5,10
```

To make life easier we could create a function to do both stages, specifying the camera number, image number, and object number to be used:

```
FUNCTION CameraOutputToObject(camno,objno,imgno)
    SET CAMERA TO IMAGE camno,imgno,512,512
    TEXTURE OBJECT objno,imgno
ENDFUNCTION
```

If you wanted to allow the dimensions of the image to be set, these could be added to the parameter list for the function.

The program in LISTING-33.10 makes use of this routine to display the output of camera 1 onto a cube placed in the grassy area in front of the castle.

LISTING-33.10

Showing Camera Output
on an Object's Surface

```
#INCLUDE "Castle.dba"
#INCLUDE "Camera.dba"

REM *** set up screen ***
SET DISPLAY MODE 1280,1024,32
AUTOCAM OFF

DrawCastle()

CreateReflectiveCube()

REM *** Create and use camera 2 ***
MAKE CAMERA 2
POSITION CAMERA 2,0,8,10
POINT CAMERA 2,0,8,200
SwitchCamera(2)

REM *** Let user control camera ***
DO
    PointCameraUsingMouse(2)
    MoveCameraUsingMouse(2,1)
LOOP

REM *** End program ***
END

FUNCTION CreateReflectiveCube()
    REM *** Make cube ***
    MAKE OBJECT CUBE 800,10
    POSITION OBJECT 800,200,5.0001,50
    REM *** Create camera inside cube ***
    MAKE CAMERA 1
    POSITION CAMERA 1,200,5.001,50
    POINT CAMERA 1, -200,5.001,50
    SET CAMERA VIEW 1, 0,0,10,10
    COLOR BACKDROP 1, RGB(255,0,0)
    BACKDROP ON 1
    REM *** Texture cube with camera's output ***
    CameraOutputToObject(1,800,222)
ENDFUNCTION

FUNCTION CameraOutputToObject(camno,objno, imgno)
    SET CAMERA TO IMAGE camno,imgno,512,1024
    TEXTURE OBJECT objno,imgno
ENDFUNCTION
```

Activity 33.31

Type in and test the program in LISTING-33.10 (*camera10.dbpro*).

Since all six sides of the cube show the same camera output, it is really only acting as a mirror on the side which points in the same direction as the camera. To create a more realistic mirror, we could show the camera's output on a plane.

Activity 33.32

In your previous program, replace the cube with a plane. The plane should be perpendicular to the grass and facing in the same direction as camera 1.

Modify the main section so that the plane rotates slowly about its y-axis.

The SET CAMERA TO OBJECT ORIENTATION Statement

We can make a camera point in the same direction as a specific object using the SET CAMERA TO OBJECT ORIENTATION statement. This statement has the format shown in FIG-33.41.

FIG-33.41 The SET CAMERA TO OBJECT ORIENTATION Statement



In the diagram:

camno

is an integer value specifying the camera involved.

objno

is an integer value specifying the ID of the object whose orientation the camera is to match.

We can make use of this statement in the last program, making camera 1 always point in the same direction as the plane. That way the illusion of a mirror is maintained. We can achieve the effect using the statement:

```
SET CAMERA TO OBJECT ORIENTATION 1,800
```

The statement needs to be executed each time the object specified in the statement moves.

Activity 33.33

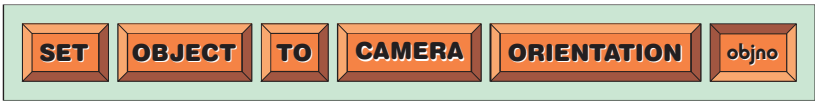
In your last program, add the line given above as the last statement within the DO..LOOP structure and observe how the image *reflected* by the plane changes as the plane rotates.

The SET OBJECT TO CAMERA ORIENTATION Statement

You can't have failed to notice that the trees along the edge of the castle's pathway look a little two dimensional and that shouldn't really be a surprise since they are nothing more than textures on a plane. However, we can hide this 2D aspect of the trees by making sure that the trees always face towards the camera. That way we never get to see them edge on. We can achieve this in DarkBASIC Pro by using the SET OBJECT TO CAMERA ORIENTATION statement which turns an object to face in the same direction as the active camera. The statement has the format shown in FIG-33.42.

FIG-33.42

The SET OBJECT TO
CAMERA
ORIENTATION Statement



In the diagram:

objno is an integer value specifying the ID of the object whose orientation is to be made to match that of the active camera.

The following function turns all the trees in the castle scene to face away from the player (hence they always have their back to the player).

```
FUNCTION OrientTrees()  
  FOR c = tree1 TO tree1 + 30  
    SET OBJECT TO CAMERA ORIENTATION c  
  NEXT c  
ENDFUNCTION
```

Activity 33. 34

Add the function given above to your last program.

Add a call to the function as your last statement in the DO..LOOP structure in the main section of the program.

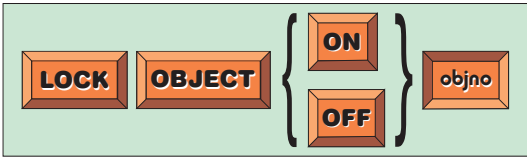
The LOCK OBJECT Statement

In First Person Shooter (FPS) games we almost always see the weapon our character is carrying. No matter how the character turns, twists, or jumps, the weapon is always there in the same position.

This effect can be toggled on or off in DarkBASIC Pro using the LOCK OBJECT statement. LOCK OBJECT ON locks a specified object to a fixed position on the screen no matter how the camera moves. To return an object to normal, use the LOCK OBJECT OFF option. The statement has the format shown in FIG-33.43.

FIG-33.43

The LOCK OBJECT
Statement



In the diagram:

ON Use this option to lock an object to the screen.

OFF Use this option to return a locked object to normal.

objno is an integer value giving the ID of the object whose position is to be locked/unlocked.

Once this command has been executed, the objected specified will stay at exactly the same spot on the screen irrespective of camera movement, but its position can still be modified using statements such as POSITION OBJECT and ROTATE OBJECT.

In LISTING-33.11 a wand (constructed from a cylinder) is locked to the screen - we'll assume our character is a wizard rather than a warrior!

A background cube gives some depth to the program and the user can move the camera using the arrow keys.

LISTING-33.11

Fixing An Object's
Screen Position

```
REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
HIDE MOUSE
CreateWand()
REM *** Create a background cube ***
MAKE OBJECT CUBE 2,10
POSITION OBJECT 2, 0,0,50
REM *** Control camera ***
DO
    CONTROL CAMERA USING ARROWKEYS 0,0.1,0.1
LOOP
REM *** End program ***
END

FUNCTION CreateWand()
    REM *** Assign ID names ***
    #CONSTANT wandobj 777
    #CONSTANT wandimg 777
    REM *** Make wand ***
    MAKE OBJECT CYLINDER wandobj,1
    LOAD IMAGE "wand.jpg",wandimg
    TEXTURE OBJECT wandobj,wandimg
    REM *** Fix wand to screen ***
    LOCK OBJECT ON wandobj
    REM *** Scale, rotate and position the wand ***
    SCALE OBJECT wandobj, 10,100,10
    XROTATE OBJECT wandobj,45
    POSITION OBJECT wandobj,0.5,-1,2
ENDFUNCTION
```

Activity 33.35

Type in and test the program given in LISTING-33.11 (*camera11.dbpro*).

Add the *CreateWand()* function to the last version of *camera10.dbpro* and place a call to the function immediately before the DO..LOOP in the main section of the program.

Test the result. Are there any problems with the mirrored plane?

There are a couple of camera statements that make use of 3D vectors. These statements are just alternatives to other camera statements we've already covered, but are included here for completeness. 3D vectors themselves are covered in a later chapter.

The SET VECTOR3 TO CAMERA POSITION Statement

We can save the coordinates of a camera to a 3D vector using the SET VECTOR3 TO CAMERA POSITION statement. This statement has the format shown in FIG-33.44.

FIG-33.44 The SET VECTOR3 TO CAMERA POSITION Statement



In the diagram:

<i>3DVector</i>	is an integer value specifying the 3D vector in which the camera's position is to be stored.
<i>camno</i>	is an integer value giving the ID of the camera whose position is to be stored.

For example, we could store the current position of camera zero in a 3D vector using the lines:

```
result = MAKE VECTOR3(1)           `Create 3D vector
SET VECTOR3 TO CAMERA POSITION 1,0
```

The SET VECTOR3 TO CAMERA ROTATION Statement

The current angles of rotation of a camera (about all three of its axes) can be stored in a single 3D vector using the SET VECTOR3 TO CAMERA ROTATION statement which has the format shown in FIG-33.45.

FIG-33.45 The SET VECTOR3 TO CAMERA ROTATION Statement



In the diagram:

<i>3DVector</i>	is an integer value specifying the 3D vector in which the camera's rotation is to be stored.
<i>camno</i>	is an integer value giving the ID of the camera whose rotation details are to be stored.

For example, we could store the current rotation details of camera zero in a 3D vector using the lines:

```
result = MAKE VECTOR3(2)           `Create 3D vector
SET VECTOR3 TO CAMERA POSITION 2,0
```

Summary

- Use SET CAMERA TO IMAGE to send a camera's output to an image object.
- An image containing a camera's output can be used to texture a 3D object.
- Use SET CAMERA TO OBJECT ORIENTATION to make a camera face in the same direction as a specified object.

- Use SET OBJECT TO CAMERA ORIENTATION to make an object face in the same direction as the active camera.
- Use LOCK OBJECT ON to fix an object position on the screen irrespective of camera movement.
- Use LOCK OBJECT OFF to return an object to normal positioning.
- Use SET VECTOR3 TO CAMERA POSITION to store a camera's coordinates in a 3D vector.
- Use SET VECTOR3 TO CAMERA ROTATION to store a camera's angles of rotation in a 3D vector.

Solutions

Activity 33.1

The main section should be coded as:

```
REM *** Include DrawCastle() function ***
#include "castle.dba"
REM *** Set screen resolution ***
SET DISPLAY MODE 1280,1024,32
DrawCastle()
WAIT KEY
POSITION CAMERA 0,0,5,50
WAIT KEY
END
```

Make sure you have copied *Castle.dba* into the same folder as this program.

The camera has moved closer to the castle so that the 3D objects now occupy the whole screen.

Activity 33.2

The program should now read:

```
REM *** Include DrawCastle() function ***
#include "castle.dba"
REM *** Set screen resolution ***
SET DISPLAY MODE 1280,1024,32
DrawCastle()
WAIT KEY
MOVE CAMERA 50
WAIT KEY
END
```

The camera moves 50 units from its starting position in the direction it is facing when a key is pressed. The jump from the starting position to the final position is instant.

Activity 33.3

The latest version of the program reads:

```
REM *** Include DrawCastle() function ***
#include "castle.dba"
REM *** Set screen resolution ***
SET DISPLAY MODE 1280,1024,32
DrawCastle()
WAIT KEY
REM *** Move camera by increments ***
FOR c = 1 TO 50
    MOVE CAMERA 1
    WAIT 50
NEXT c
WAIT KEY
END
```

Activity 33.4

```
REM *** Include DrawCastle() function ***
#include "castle.dba"
REM *** Set screen resolution ***
SET DISPLAY MODE 1280,1024,32
DrawCastle()
WAIT KEY
REM *** Point camera in direction of move **
POINT CAMERA -25,5,700
REM *** Move camera by increments ***
FOR c = 1 TO 100
    MOVE CAMERA 1
    WAIT 50
```

```
NEXT c
WAIT KEY
END
```

To make the camera move right up to the castle wall, change the FOR loop to read

```
FOR c = 1 TO 750
```

Activity 33.5

```
REM *** Include DrawCastle() function ***
#include "castle.dba"
REM *** Set screen resolution ***
SET DISPLAY MODE 1280,1024,32
DrawCastle()
WAIT KEY
REM *** Turn camera by increments ***
FOR angle = -45 TO 45
    ROTATE CAMERA angle,0,0
    WAIT 10
NEXT angle
REM *** End program ***
WAIT KEY
END
```

To rotate the camera about the y-axis change the ROTATE CAMERA line to

```
ROTATE CAMERA 0,angle,0
```

Rotation about the z-axis requires the line

```
ROTATE CAMERA 0,0,angle
```

Activity 33.6

```
REM *** Include DrawCastle() function ***
#include "castle.dba"
REM *** Set screen resolution ***
SET DISPLAY MODE 1280,1024,32
DrawCastle()
REM *** Start with zero rotation ***
ROTATE CAMERA 0,0,0
DO
    REM *** Rotate camera using keys ***
    SELECT INKEY$( )
        CASE "z"
            TURN CAMERA LEFT 1
        ENDCASE
        CASE "x"
            TURN CAMERA RIGHT 1
        ENDCASE
        CASE "'"
            PITCH CAMERA UP 1
        ENDCASE
        CASE "/"
            PITCH CAMERA DOWN 1
        ENDCASE
        CASE "n"
            ROLL CAMERA LEFT 1
        ENDCASE
        CASE "m"
            ROLL CAMERA RIGHT 1
        ENDCASE
    ENDSELECT
LOOP
REM *** End program ***
END
```


Activity 33.7

```
REM *** Include DrawCastle() function ***
#INCLUDE "castle.dba"
REM *** Set screen resolution ***
SET DISPLAY MODE 1280,1024,32
POSITION CAMERA 0,0,0
POINT CAMERA 0,0,0
DrawCastle()
ROTATE CAMERA 0,0,0
DO
    REM *** Rotate camera on key press ***
    SELECT INKEY$( )
        CASE "z"
            TURN CAMERA LEFT 0.1
        ENDCASE
        CASE "x"
            TURN CAMERA RIGHT 0.1
        ENDCASE
        CASE " "
            PITCH CAMERA UP 0.1
        ENDCASE
        CASE "/"
            PITCH CAMERA DOWN 0.1
        ENDCASE
        CASE "n"
            ROLL CAMERA LEFT 0.1
        ENDCASE
        CASE "m"
            ROLL CAMERA RIGHT 0.1
        ENDCASE
    ENDSELECT
    REM *** Get camera angles of rotation ***
    xang# = CAMERA ANGLE X(0)
    yang# = CAMERA ANGLE Y(0)
    zang# = CAMERA ANGLE Z(0)
    REM *** Display angles ***
    SET CURSOR 100,100
    PRINT "Angle about x-axis: ",xang#
    SET CURSOR 100,120
    PRINT "Angle about y-axis: ",yang#
    SET CURSOR 100,140
    PRINT "Angle about z-axis: ",zang#
LOOP
REM *** End program ***
END
```

Activity 33.8

```
REM *** Include DrawCastle() function ***
#INCLUDE "castle.dba"
REM *** Set screen resolution ***
SET DISPLAY MODE 1280,1024,32
POSITION CAMERA 0,0,0
POINT CAMERA 0,0,0
DrawCastle()
ROTATE CAMERA 0,0,0
REM *** Set camera output area ***
SET CAMERA VIEW 0,0,1279,319
DO
    REM *** Rotate camera on key press ***
    SELECT INKEY$( )
        CASE "z"
            TURN CAMERA LEFT 0.1
        ENDCASE
        CASE "x"
            TURN CAMERA RIGHT 0.1
        ENDCASE
        CASE " "
            PITCH CAMERA UP 0.1
        ENDCASE
        CASE "/"
            PITCH CAMERA DOWN 0.1
        ENDCASE
        CASE "n"
            ROLL CAMERA LEFT 0.1
        ENDCASE
    ENDSELECT
```

```
        CASE "m"
            ROLL CAMERA RIGHT 0.1
        ENDCASE
    ENDSELECT
    REM *** Get camera angles of rotation ***
    xang# = CAMERA ANGLE X(0)
    yang# = CAMERA ANGLE Y(0)
    zang# = CAMERA ANGLE Z(0)
    REM *** Display angles ***
    SET CURSOR 100,400
    PRINT "Angle about x-axis: ",xang#
    SET CURSOR 100,420
    PRINT "Angle about y-axis: ",yang#
    SET CURSOR 100,440
    PRINT "Angle about z-axis: ",zang#
LOOP
REM *** End program ***
END
```

Since the camera does not output to the black area of the screen, that area is not continually redrawn and so we have the same problem with overwriting screen areas that we encountered way back in Chapter 2. We can solve this by changing the display lines to read:

```
SET TEXT OPAQUE
SET CURSOR 100,400
PRINT "
SET CURSOR 100,400
PRINT "Angle about x-axis: ",xang#
SET CURSOR 100,420
PRINT "
SET CURSOR 100,420
PRINT "Angle about y-axis: ",yang#
SET CURSOR 100,440
PRINT "
SET CURSOR 100,440
PRINT "Angle about z-axis: ",zang#
```

Activity 33.9

```
REM *** Include DrawCastle() function ***
#INCLUDE "castle.dba"
REM *** Set screen resolution ***
SET DISPLAY MODE 1280,1024,32
POSITION CAMERA 0,0,0
POINT CAMERA 0,0,0
DrawCastle()
ROTATE CAMERA 0,0,0
REM *** Set camera output area ***
SET CAMERA VIEW 0,0,1279,319
REM *** Set aspect ratio ***
SET CAMERA ASPECT 1280/320
DO
    REM *** Rotate camera on key press ***
    SELECT INKEY$( )
        CASE "z"
            TURN CAMERA LEFT 0.1
        ENDCASE
        CASE "x"
            TURN CAMERA RIGHT 0.1
        ENDCASE
        CASE " "
            PITCH CAMERA UP 0.1
        ENDCASE
        CASE "/"
            PITCH CAMERA DOWN 0.1
        ENDCASE
        CASE "n"
            ROLL CAMERA LEFT 0.1
        ENDCASE
        CASE "m"
            ROLL CAMERA RIGHT 0.1
        ENDCASE
    ENDSELECT
```

```

REM *** Get camera angles of rotation ***
xang# = CAMERA ANGLE X(0)
yang# = CAMERA ANGLE Y(0)
zang# = CAMERA ANGLE Z(0)
REM *** Display angles ***
SET TEXT OPAQUE
SET CURSOR 100,400
PRINT "
SET CURSOR 100,400
PRINT "Angle about x-axis: ",xang#
SET CURSOR 100,420
PRINT "
SET CURSOR 100,420
PRINT "Angle about y-axis: ",yang#
SET CURSOR 100,440
PRINT "
SET CURSOR 100,440
PRINT "Angle about z-axis: ",zang#
LOOP
REM *** End program ***
END

```

```

↵CAMERA POSITION Y(),"","
↵CAMERA POSITION Z(),"")"
LOOP
REM *** End program ***
END

```

The cube becomes invisible because the camera is placed at its centre (0,0,0).

Activity 33.13

No solution required.

Activity 33.14

```

#include "Castle.dba"
#CONSTANT artifact 200
#CONSTANT arttexture 200
REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP RGB(64,0,128)
BACKDROP ON
REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 0,8,10
REM *** Draw castle ***
DrawCastle()
REM *** Show falling object ***
FallingObject()
REM *** End program ***
WAIT KEY
END

```

```

FUNCTION FallingObject()
REM *** Create textured trans. sphere ***
MAKE OBJECT SPHERE artifact,10
LOAD IMAGE "lattice.bmp",arttexture
TEXTURE OBJECT artifact,arttexture
SET OBJECT TRANSPARENCY artifact,1
REM *** Position artifact out in sky ***
POSITION OBJECT artifact, -200,300,900
REM *** Point object towards ground ***
POINT OBJECT artifact,300,0,30
REM *** Move object, stopping when ***
REM *** it hits the ground ***
DO
MOVE OBJECT artifact,8
REM *** Train camera on object ***
POINT CAMERA OBJECT POSITION
↵X(artifact),
↵OBJECT POSITION Y(artifact),
↵OBJECT POSITION Z(artifact),
IF OBJECT POSITION Y(artifact) <= 2
EXIT
ENDIF
LOOP
ENDFUNCTION

```

Activity 33.10

No solution required.

Activity 33.11

```

#include "Castle.dba"
REM ***Set screen resolution ***
SET DISPLAY MODE 1280,1024,32
REM *** Draw the castle ***
DrawCastle()
REM *** Position camera ***
POSITION CAMERA 0,8,0
REM *** Set the camera screen area ***
SET CAMERA VIEW 0,0,1279,639
REM *** Set the aspect ratio ***
SET CAMERA ASPECT 2/1
REM *** Point camera at turret ***
POINT CAMERA -10,75,200
REM *** Change camera range ***
SET CAMERA RANGE 50,500
REM *** Zoom in on turret ***
WAIT KEY
FOR angle = 180 to 5 STEP -1
SET CAMERA FOV angle
WAIT 50
NEXT angle
REM *** Move camera towards turret ***
WAIT KEY
SET CAMERA FOV 55
POINT CAMERA -10,75,200
FOR move = 1 TO 580
MOVE CAMERA 1
WAIT 20
NEXT move
REM *** End program ***
WAIT KEY
END

```

Activity 33.12

```

REM *** Switch off auto camera placing ***
AUTOCAM OFF
REM *** Set screen resolution ***
SET DISPLAY MODE 1280,1024,32
REM *** Get size of cube ***
INPUT "Enter size of cube (2 to 100):",size
REM *** Create cube ***
MAKE OBJECT CUBE 1, size
REM *** Display position of camera ***
DO
SET CURSOR 100,100
PRINT "(, CAMERA POSITION X(),"","

```

Activity 33.15

The final version of the *FallingObject()* function should read as follows:

```

FUNCTION FallingObject()
REM *** Create textured trans. sphere ***
MAKE OBJECT SPHERE artifact,10
LOAD IMAGE "lattice.bmp",arttexture
TEXTURE OBJECT artifact,arttexture
SET OBJECT TRANSPARENCY artifact,1
REM *** Position artifact out in sky ***
POSITION OBJECT artifact, -200,300,900
REM *** Point object towards ground ***
POINT OBJECT artifact,300,0,30
REM *** Move object, stopping when ***

```

```

REM *** it hits the ground ***
DO
    MOVE OBJECT artifact,8
    REM *** Make camera follow object ***
    SET CAMERA TO FOLLOW OBJECT POSITION
    ↵X(artifact),
    ↵OBJECT POSITION Y(artifact),
    ↵OBJECT POSITION Z(artifact),
    ↵90,100,20,1.0,1
    IF OBJECT POSITION Y(artifact) <= 2
        EXIT
    ENDFIF
LOOP
ENDFUNCTION

```

```

XnewAngle# = XcurrentAngle#
↵+ MOUSEMOVEY()*0.1
IF XnewAngle# < -90
    XnewAngle# = -90
ELSE
    IF XnewAngle# > 90
        XnewAngle# = 90
    ENDIF
ENDIF
YnewAngle# = WRAPVALUE(YcurrentAngle#
↵+ MOUSEMOVEX()*0.1)
REM *** Rotate camera ***
XROTATE CAMERA camno,XnewAngle#
YROTATE CAMERA camno,YnewAngle#
ENDFUNCTION

```

Activity 33.16

The SET CAMERA TO FOLLOW line in your program should read

```

SET CAMERA TO FOLLOW
↵OBJECT POSITION X(artifact),
↵OBJECT POSITION Y(artifact),
↵OBJECT POSITION Z(artifact),
↵90,100,20,100.0,1

```

The camera now swings smoothly into place from its original position, rather than jumping to the new position abruptly.

With a *smooth* value of 20 the jump between the original position and the new position is faster than with a setting of 100.

The camera now moves more slowly in reaction to mouse movements.

Activity 33.22

The updated program is now:

```

#include "Castle.dba"
REM *** set up screen ***
SET DISPLAY MODE 1280,1024,32
AUTOCAM OFF
DrawCastle()
REM *** Re-position camera ***
POSITION CAMERA 0,8,10
POINT CAMERA 0,8,200
DO
    PointCameraUsingMouse(0)
    CONTROL CAMERA USING ARROWKEYS 0,0.5,0
LOOP
REM *** End program ***
END

REM *** Rotates the active camera ***
FUNCTION PointCameraUsingMouse(camno)
    REM *** Get current camera angles ***
    XcurrentAngle# = CAMERA ANGLE X(camno)
    YcurrentAngle# = CAMERA ANGLE Y(camno)
    XnewAngle# = XcurrentAngle#
    ↵+ MOUSEMOVEY()*0.1
    IF XnewAngle# < -90
        XnewAngle# = -90
    ELSE
        IF XnewAngle# > 90
            XnewAngle# = 90
        ENDIF
    ENDIF
    YnewAngle# = WRAPVALUE(YcurrentAngle#
    ↵+ MOUSEMOVEX()*0.1)
    REM *** Rotate camera ***
    XROTATE CAMERA camno,XnewAngle#
    YROTATE CAMERA camno,YnewAngle#
ENDFUNCTION

```

Activity 33.17

No solution required.

Activity 33.18

Yes. The camera can move straight through solid objects.

Activity 33.19

No solution required.

Activity 33.20

The camera turns too quickly to gain accurate control.

Activity 33.21

```

#include "Castle.dba"
REM *** set up screen ***
SET DISPLAY MODE 1280,1024,32
AUTOCAM OFF
DrawCastle()
REM *** Re-position camera ***
POSITION CAMERA 0,8,10
POINT CAMERA 0,8,200
DO
    PointCameraUsingMouse(0)
LOOP
REM *** End program ***
END
REM *** Rotates the active camera ***
FUNCTION PointCameraUsingMouse(camno)
    REM *** Get current camera angles ***
    XcurrentAngle# = CAMERA ANGLE X(camno)
    YcurrentAngle# = CAMERA ANGLE Y(camno)

```

The camera can no longer point up or down.

Activity 33.23

```

#include "Castle.dba"
REM *** set up screen ***
SET DISPLAY MODE 1280,1024,32
AUTOCAM OFF
DrawCastle()
REM *** Re-position camera ***
POSITION CAMERA 0,8,10
POINT CAMERA 0,8,200
DO
    PointCameraUsingMouse(0)
    MoveCameraUsingMouse(0,1)
LOOP

```

```

REM *** End program ***
END

REM *** Rotates the active camera ***
FUNCTION PointCameraUsingMouse(camno)
  REM *** Get current camera angles ***
  XcurrentAngle# = CAMERA ANGLE X(camno)
  YcurrentAngle# = CAMERA ANGLE Y(camno)
  XnewAngle# = XcurrentAngle#
  ⌘+ MOUSEMOVEVY()*0.1
  IF XnewAngle# < -90
    XnewAngle# = -90
  ELSE
    IF XnewAngle# > 90
      XnewAngle# = 90
    ENDIF
  ENDIF
  YnewAngle# = WRAPVALUE(YcurrentAngle#
  ⌘+ MOUSEMOVEVX()*0.1)
  REM *** Rotate camera ***
  XROTATE CAMERA camno,XnewAngle#
  YROTATE CAMERA camno,YnewAngle#
ENDFUNCTION

FUNCTION MoveCameraUsingMouse
  ⌘(camno,distance#)
  IF MOUSECLICK() = 1
    MOVE CAMERA camno,distance#
  ELSE
    IF MOUSECLICK() = 2
      MOVE CAMERA camno,-distance#
    ENDIF
  ENDIF
ENDFUNCTION

```

Activity 33.24

```

#include "Castle.dba"
REM *** set up screen ***
SET DISPLAY MODE 1280,1024,32
AUTOCAM OFF
DrawCastle()
REM *** Re-position camera ***
POSITION CAMERA 0,8,10
POINT CAMERA 0,8,200
AUTOMATIC CAMERA COLLISION 0,2,1
DO
  PointCameraUsingMouse(0)
  MoveCameraUsingMouse(0,1)
LOOP
REM *** End program ***
END

```

```

REM *** Rotates the active camera ***
FUNCTION PointCameraUsingMouse(camno)
  REM *** Get current camera angles ***
  XcurrentAngle# = CAMERA ANGLE X(camno)
  YcurrentAngle# = CAMERA ANGLE Y(camno)
  XnewAngle# = XcurrentAngle#
  ⌘+ MOUSEMOVEVY()*0.1
  IF XnewAngle# < -90
    XnewAngle# = -90
  ELSE
    IF XnewAngle# > 90
      XnewAngle# = 90
    ENDIF
  ENDIF
  YnewAngle# = WRAPVALUE(YcurrentAngle#
  ⌘+ MOUSEMOVEVX()*0.1)
  REM *** Rotate camera ***
  XROTATE CAMERA camno,XnewAngle#
  YROTATE CAMERA camno,YnewAngle#
ENDFUNCTION

```

```

FUNCTION MoveCameraUsingMouse
  ⌘(camno, distance)
  REM *** Save angle about x axis ***

```

```

Xangle# = CAMERA ANGLE X(camno)
REM *** Point camera along ground ***
XROTATE CAMERA camno,0
IF MOUSECLICK() = 1
  MOVE CAMERA camno,distance
ELSE
  IF MOUSECLICK() = 2
    MOVE CAMERA camno,-distance
  ENDIF
ENDIF
REM *** Restore angle about x-axis ***
XROTATE CAMERA camno,Xangle#
REM *** Keep camera within area ***
x# = CAMERA POSITION X(0)
z# = CAMERA POSITION Z(0)
IF x#<-50
  x# = -50
ELSE
  IF x#>500
    x#=500
  ENDIF
ENDIF
IF z# < 0
  z# = 0
ELSE
  IF z# > 700
    z# = 700
  ENDIF
ENDIF
POSITION CAMERA 0,x#,
⌘CAMERA POSITION Y(0),z#
ENDFUNCTION

```

Activity 33.25

No solution required.

Activity 33.26

The grass and path are not visible because camera 1 has been created at position (0,0,0) and is therefore edge on to those two planes.

Activity 33.27

No solution required.

Activity 33.28

```

#include "Castle.dba"
REM *** set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP RGB(64,0,128)
BACKDROP ON
AUTOCAM OFF
DrawCastle()
REM *** Re-position the default camera ***
POSITION CAMERA -20,8,10
POINT CAMERA 0,8,200
REM *** Make new camera ***
WAIT KEY
MAKE CAMERA 1
REM *** Make background blue ***
COLOR BACKDROP 1, RGB(0,0,255)
REM *** End program ***
WAIT KEY
END

```

Activity 33.29

No solution required.

Activity 33.30

```
#INCLUDE "Castle.dba"
#INCLUDE "Camera.dba"
REM *** set up screen ***
SET DISPLAY MODE 1280,1024,32
AUTOCAM OFF
DrawCastle()
REM *** Re-position camera ***
POSITION CAMERA 0,8,10
POINT CAMERA 0,8,200
REM *** Create a second camera ...***
MAKE CAMERA 1
POSITION CAMERA 1,50,250,900
POINT CAMERA 1,200,0,0
REM *** ... with its own output area ***
SET CAMERA VIEW 1, 800,0,1279,300
REM *** Control camera movement ***
DO
    PointCameraUsingMouse(0)
    MoveCameraUsingMouse(0,0.5)
    TURN CAMERA LEFT 1,0.1
LOOP
REM *** End program ***
WAIT KEY
END
```

Remember to copy *Camera.dba* to the current folder.

Activity 33.31

No solution required.

Activity 33.32

```
#INCLUDE "Castle.dba"
#INCLUDE "Camera.dba"
REM *** set up screen ***
SET DISPLAY MODE 1280,1024,32
AUTOCAM OFF
DrawCastle()
CreateReflectivePlane()
REM *** Create and use camera 2 ***
MAKE CAMERA 2
POSITION CAMERA 2,0,8,10
POINT CAMERA 2,0,8,200
SwitchCamera(2)
REM *** Let user control camera ***
DO
    PointCameraUsingMouse(2)
    MoveCameraUsingMouse(2,1)
    TURN OBJECT LEFT 800,0.1
LOOP
REM *** End program ***
END

FUNCTION CreateReflectivePlane()
    REM *** Make plane ***
    MAKE OBJECT PLAIN 800,10,10
    POSITION OBJECT 800,200,5.001,50
    POINT OBJECT 800,-200,5,50
    REM *** Create camera in mid-plane***
    MAKE CAMERA 1
    POSITION CAMERA 1,200,5,50
    POINT CAMERA 1, -200,5,50
    SET CAMERA VIEW 1, 0,0,10,10
    COLOR BACKDROP 1, RGB(255,0,0)
    BACKDROP ON 1
    REM *** Camera's output as texture ***
    CameraOutputToObject(1,800,222)
ENDFUNCTION

FUNCTION CameraOutputToObject
    (camno,objno, imgno)
    SET CAMERA TO IMAGE camno,imgno,512,1024
    TEXTURE OBJECT objno,imgno
```

ENDFUNCTION

Activity 33.33

The DO..LOOP in the main section should now read:

```
DO
    PointCameraUsingMouse(2)
    MoveCameraUsingMouse(2,1)
    TURN OBJECT LEFT 800,0.1
    SET CAMERA TO OBJECT ORIENTATION 1,800
LOOP
```

Activity 33.34

```
#INCLUDE "Castle.dba"
#INCLUDE "Camera.dba"
REM *** set up screen ***
SET DISPLAY MODE 1280,1024,32
AUTOCAM OFF
DrawCastle()
CreateReflectivePlane()
REM *** Create and use camera 2 ***
MAKE CAMERA 2
POSITION CAMERA 2,0,8,10
POINT CAMERA 2,0,8,200
SwitchCamera(2)
REM *** Let user control camera ***
DO
    PointCameraUsingMouse(2)
    MoveCameraUsingMouse(2,1)
    TURN OBJECT LEFT 800,0.1
    SET CAMERA TO OBJECT ORIENTATION 1,800
    OrientTrees()
LOOP
REM *** End program ***
END

FUNCTION CreateReflectivePlane()
    REM *** Make cube ***
    MAKE OBJECT PLAIN 800,10,10
    POSITION OBJECT 800,200,5.001,50
    POINT OBJECT 800,-200,5,50
    REM *** Camera in middle of plane***
    MAKE CAMERA 1
    POSITION CAMERA 1,200,5,50
    POINT CAMERA 1, -200,5,50
    SET CAMERA VIEW 1, 0,0,10,10
    COLOR BACKDROP 1, RGB(255,0,0)
    BACKDROP ON 1
    REM *** Texture with camera's output ***
    CameraOutputToObject(1,800,222)
ENDFUNCTION

FUNCTION CameraOutputToObject
    (camno,objno, imgno)
    SET CAMERA TO IMAGE camno,imgno,512,1024
    TEXTURE OBJECT objno,imgno
ENDFUNCTION

FUNCTION OrientTrees()
    FOR c = treel TO treel + 30
        SET OBJECT TO CAMERA ORIENTATION c
    NEXT c
ENDFUNCTION
```

Activity 33.35

```
#INCLUDE "Castle.dba"
#INCLUDE "Camera.dba"
REM *** set up screen ***
SET DISPLAY MODE 1280,1024,32
AUTOCAM OFF
DrawCastle()
CreateReflectivePlane()
```

```

REM *** Create and use camera 2 ***
MAKE CAMERA 2
POSITION CAMERA 2,0,8,10
POINT CAMERA 2,0,8,200
SwitchCamera(2)
REM *** Let user control camera ***
CreateWand()
DO
    PointCameraUsingMouse(2)
    MoveCameraUsingMouse(2,1)
    TURN OBJECT LEFT 800,0.1
    SET CAMERA TO OBJECT ORIENTATION 1,800
    OrientTrees()
LOOP
REM *** End program ***
END

FUNCTION CreateReflectivePlane()
    REM *** Make cube ***
    MAKE OBJECT PLAIN 800,10,10
    POSITION OBJECT 800,200,5.001,50
    POINT OBJECT 800,-200,5,50
    REM *** Camera in middle of plane***
    MAKE CAMERA
    POSITION CAMERA 1,200,5,50
    POINT CAMERA 1, -200,5,50
    SET CAMERA VIEW 1, 0,0,10,10
    COLOR BACKDROP 1, RGB(255,0,0)
    BACKDROP ON 1
    REM *** Texture with camera's output ***
    CameraOutputToObject(1,800,222)
ENDFUNCTION

FUNCTION CameraOutputToObject
↳ (camno,objno, imgno)
    SET CAMERA TO IMAGE camno,imgno,512,1024
    TEXTURE OBJECT objno,imgno
ENDFUNCTION

FUNCTION OrientTrees()
    FOR c = tree1 TO tree1 + 30
        SET OBJECT TO CAMERA ORIENTATION c
    NEXT c
ENDFUNCTION

FUNCTION CreateWand()
    REM *** Assign ID names ***
    #CONSTANT wandobj 777
    #CONSTANT wandimg 777
    REM *** Make wand ***
    MAKE OBJECT CYLINDER wandobj,1
    LOAD IMAGE "wand.jpg",wandimg
    TEXTURE OBJECT wandobj,wandimg
    REM *** Fix wand to screen ***
    LOCK OBJECT ON wandobj
    REM *** Scale, rotate & position wand ***
    SCALE OBJECT wandobj, 10,100,10
    XROTATE OBJECT wandobj,45
    POSITION OBJECT wandobj,0.5,-1,2
ENDFUNCTION

```

The wand also appears on the output of any other cameras being used.

Changing How Objects React to Light

Creating Coloured Lights

Fog

Modifying a Light's Brightness

Positioning Lights

Types of Lighting

Lighting

Introduction

By adding the correct lighting, we can transform a scene from something ordinary into a place with atmosphere. Lighting can alter a location from appearing bright and happy to somewhere dark and sinister. Take your time thinking about lighting and transform your game!

Types of Lighting

There are several types of lighting, each of which are described below.

Ambient Lighting

An ambient light is one with no obvious source and no obvious direction. Ambient light is everywhere and appears to come from every direction. It's the sort of light you might get on a heavily overcast day in the middle of winter.

Point Lighting

Point lighting is light that originates from a specific point in space and casts light equally in all directions. The further away we move from the source, the duller the light becomes. This is similar to a naked light bulb.

Spot Lighting

A spot light originates from a point in space and the light itself shines in a specific direction creating a cone-shaped light. The angle over which the light shines can be set and the light consists of a primary and secondary area, with the secondary area containing a duller light than the primary area. Again the light diminishes the further we travel from the source. We can see examples of spotlights in old war movies, the spotlights shining into the skies searching for enemy bombers.

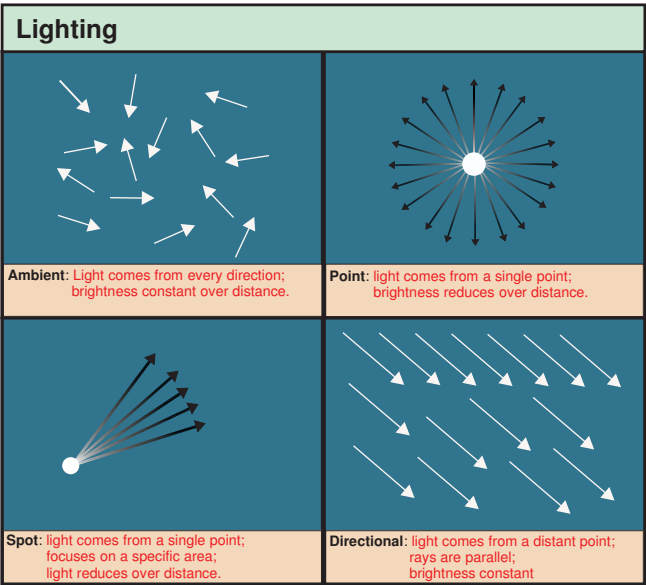
Directional Lighting

A directional light originates from an infinitely far point, but its brightness does not diminish over distance. Sunshine on a cloudless day is the obvious example for this type of lighting.

Fig-34.1 shows the basic characteristics of each type of lighting.

FIG-34.1

Lighting

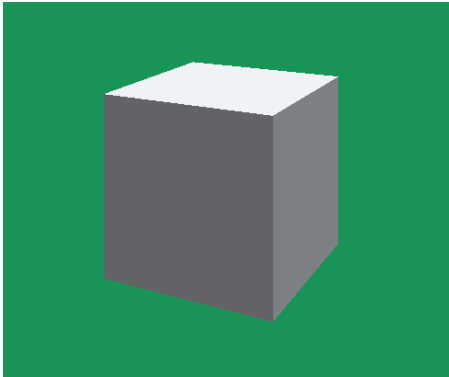


Lighting in DarkBASIC Pro

When you run a 3D program in DarkBASIC Pro, two light sources are automatically included: a directional light and an ambient light. If we examine the image shown in FIG-34.2, we can see that the amount of shadow on each side of the cube gives an indication of the position of the directional light.

FIG-34.2

Shading Caused by the Default Directional Light



Every light source in DarkBASIC Pro must be allocated an ID number (only the ambient light is exempt - it has no ID value). The default directional light, created automatically, is assigned the ID number zero.

The HIDE LIGHT Statement

An existing light can be switched off using the HIDE LIGHT statement which has the format shown in FIG-34.3.

FIG-34.3

The HIDE LIGHT Statement



In the diagram:

lightno

is an integer value giving the ID of the light to be turned off.

For example, the statement

```
HIDE LIGHT 0
```

would switch off the default directional light, leaving only the weak ambient light.

The program in LISTING-34.1 shows a cube with default lighting. By pressing any key, the default directional light is switched off, leaving only the ambient light.

LISTING-34.1

Switching of the Default Directional Light

```
REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP RGB(0,200,0)
BACKDROP ON

REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA -10,10,20
```

continued on next page

LISTING-34.1
(continued)

Switching of the Default
Directional Light

```
POINT CAMERA 0,0,0

REM *** Make and position cube ***
MAKE OBJECT CUBE 2,5
POSITION OBJECT 2,0,2.5,0

REM *** Hide directional light ***
WAIT KEY
HIDE LIGHT 0

REM *** End program ***
WAIT KEY
END
```

Activity 34.1

Type in and test the program given in LISTING-34.1 (*lights01.dbpro*).

The SHOW LIGHT Statement

A light which has been switched off using the HIDE LIGHT statement can be switched back on using the SHOW LIGHT statement. This statement has the format shown in FIG-34.4.

FIG-34.4

The SHOW LIGHT
Statement



In the diagram:

lightno

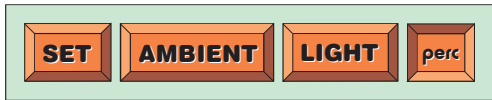
is an integer value giving the ID of the light to be turned on.

The SET AMBIENT LIGHT Statement

The brightness of the ambient light can be set using the SET AMBIENT LIGHT statement which has the format shown in FIG-34.5.

FIG-34.5

The SET AMBIENT
LIGHT Statement



In the diagram:

perc

is an integer value giving the percentage setting for the default ambient light. A value of zero will switch off the ambient light, while a value of 100 will give maximum brightness. The default ambient setting is 25.

For example, we could switch off the ambient light using the statement:

```
SET AMBIENT LIGHT 0
```

Activity 34.2

Modify your previous program so that the ambient light level gradually changes from an initial setting of 100 down to 0.

The COLOR AMBIENT LIGHT Statement

The color of the ambient light can also be set using the COLOR AMBIENT LIGHT statement which has the format shown in FIG-34.6.

FIG-34.6

The COLOR AMBIENT LIGHT Statement



In the diagram:

colour

is an integer value giving the colour setting for the ambient light. Often this value will be supplied by a call to the RGB statement.

For example, we could change the ambient light to red using the line:

```
COLOR AMBIENT LIGHT RGB(255,0,0)
```

Activity 34.3

Modify the last program to use a blue ambient light.

The MAKE LIGHT Statement

A new light source can be created using the MAKE LIGHT statement which has the format shown in FIG-34.7.

FIG-34.7

The MAKE LIGHT Statement



In the diagram:

lightno

is an integer value giving the ID of the light to be created. No two lights can be allocated the same ID value.

Like any other 3D object, lights are initially positioned at (0,0,0). When first created, every light is a white, point light.

In LISTING-34.2 the default directional light is switched off and a new light created. The effect of the new light can be seen on the side of the cube displayed by the program.

LISTING-34.2

Adding a new Light

```
REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP RGB(20,20,20)
BACKDROP ON

REM *** Use programmed screen updating ***
SYNC ON

REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA -10,-3,20
POINT CAMERA 0,0,0
```

continued on next page

LISTING-34.2
(continued)

Adding a new Light

SYNC is used so that the PRINT output remains on the screen between key presses.

```
REM *** Make and position cubes ***
MAKE OBJECT CUBE 2,5
POSITION OBJECT 2,0,2.5,-10
MAKE OBJECT CUBE 3,5
POSITION OBJECT 3,0,10,0
SYNC:SYNC

REM *** Switch off directional light ***
WAIT KEY
HIDE LIGHT 0
SYNC
PRINT "Directional light off"
SYNC

REM *** Create new light ***
WAIT KEY
MAKE LIGHT 1
SYNC
PRINT "New light created"
SYNC

REM *** End program ***
WAIT KEY
END
```

Activity 34.4

Type in and test the program in LISTING-34.2 (*lights02.dbpro*).

The DELETE LIGHT Statement

Any existing light can be deleted from RAM using the DELETE LIGHT statement which has the format shown in FIG-34.8.

FIG-34.8

The DELETE LIGHT Statement



In the diagram:

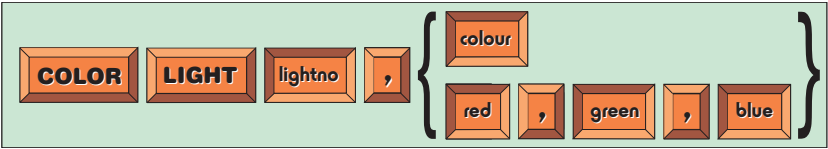
lightno is an integer value giving the ID of the light to be deleted.

The COLOR LIGHT Statement

By default lights cast a white light, but this can be changed to another colour using the COLOR LIGHT statement which has the format shown in FIG-34.9.

FIG-34.9

The COLOR LIGHT Statement



In the diagram:

lightno is an integer value giving the ID of the light whose colour is to be modified.

colour is an integer value specifying the colour of the light to be produced.

red,green,blue

is a set of integer values giving the red, green and blue component values of the required colour.

Activity 34.5

Modify your previous program so that the added light is yellow.

The POSITION LIGHT Statement

A light can be moved to a new position using the POSITION LIGHT statement. The format of this statement is given in FIG-34.10.

FIG-34.10

The POSITION LIGHT Statement



In the diagram:

lightno

is an integer value giving the ID of the light to be moved.

x,y,z

is a set of real values specifying the new position of the light.

For example, we could move light 1 to position (2,3,4) using the line:

```
POSITION LIGHT 1,2,3,4
```

Activity 34.6

In your previous program, make light 1 travel slowly away from position (0,0,0) by decrementing its y-ordinate value (use a DO..LOOP structure so the light continues to move indefinitely). Remember to use SYNC to update the screen.

The SET LIGHT RANGE Statement

Spot and point lights only illuminate a limited volume, with the light levels reducing as we move away from the source. Just how far the light's brightness extends can be set using the SET LIGHT RANGE statement which has the format given in FIG-34.11.

FIG-34.11

The SET LIGHT RANGE Statement



In the diagram:

lightno

is an integer value giving the ID of the light for which the distance the light penetrates is to be set.

distance

is a real number giving the distance in units over which the light is effective.

For example, we could set the light from light 1 to fade to complete darkness at a

distance of 10 units from the source using the line:

```
SET LIGHT RANGE 1,10
```

Activity 34.7

In your previous program, reduce the range of light 1 to 50 and observe how this affects the time it takes for the light on the cubes to fade.

The SET SPOT LIGHT Statement

Although every new light starts off as a point light, it can be changed to other types. An existing light can be changed to a spotlight using the SET SPOT LIGHT statement whose format is shown in FIG-34.12.

FIG-34.12

The SET SPOT LIGHT Statement



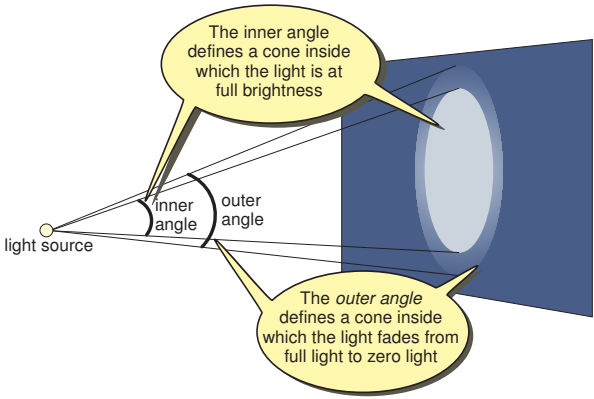
In the diagram:

- lightno* is an integer value giving the ID of the light to be changed to a spotlight.
- innerangle* is a real number giving the angle defining the cone of full light.
- outerangle* is a real number giving the angle defining the cone of decreasing light.

A visual representation of a spot light is shown in FIG-34.13.

FIG-34.13

Spot Light Explained



The SET DIRECTIONAL LIGHT Statement

We can change a light to a directional light using the SET DIRECTIONAL LIGHT statement which has the format shown in FIG-34.14.

FIG-34.14

The SET DIRECTIONAL LIGHT Statement



In the diagram:

<i>lightno</i>	is an integer value giving the ID of the light to be changed to a directional light.
<i>x,y,z</i>	is a set of real numbers giving the position to which the light is directed. This assumes the light source is currently positioned at (0,0,0).

Unlike point and spot lights, a directional light has no specific location point. Light comes from an undefined point with all rays parallel. However, in order to easily specify a direction for these rays, when a directional light is created it is assumed to be at position (0,0,0). The direction of the light emitted is then parallel to the line drawn between (0,0,0) and (x,y,z) - this last point being given in the SET DIRECTIONAL LIGHT statement. For example, we could transform an existing light 1 to a directional light and have its rays shining from directly above using the line:

```
SET DIRECTIONAL LIGHT 1,0,-1,0
```

Note that the length of the line between (0,0,0) and the point given when creating the light has no effect on the brightness of the light. Hence, the line

```
SET DIRECTIONAL LIGHT 1, 0,-30,0
```

creates light coming from exactly the same direction and brightness as the previous statement.

The SET POINT LIGHT Statement

A light can be changed back to a point light using the SET POINT LIGHT statement which has the format shown in FIG-34.15

FIG-34.15

The SET POINT LIGHT Statement



In the diagram:

<i>lightno</i>	is an integer value giving the ID of the light to be changed to a point light.
<i>x,y,z</i>	is a set of real numbers giving the new position of the light.

The POINT LIGHT Statement

NOTE: This statement has nothing to do with point light types!

Spotlights and directional lights can both be made to point in a specific direction using the POINT LIGHT statement which has the format shown in FIG-34.16.

FIG-34.16

The POINT LIGHT Statement



In the diagram:

<i>lightno</i>	is an integer value giving the ID of the light to be pointed to a specific spot.
----------------	--

x,y,z

is a set of real numbers giving the point in space to which the light is to be directed.

The program in LISTING-34.3 shines a spot light onto a sphere and then moves the light gradually away from the sphere.

LISTING-34.3

Directing a Spot Light

```
REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP RGB(20,20,20)
BACKDROP ON
REM *** Use programmed screen updating ***
SYNC ON

REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 0,0,20
POINT CAMERA 0,0,0

REM *** Make and position sphere ***
MAKE OBJECT SPHERE 2,20,150,150
SYNC:SYNC

REM *** Switch off directional light ***
WAIT KEY
HIDE LIGHT 0
SYNC
PRINT "Directional light off - press any key"
SYNC

REM *** Create spot light ***
WAIT KEY
MAKE LIGHT 1
SET SPOT LIGHT 1,40,60
POSITION LIGHT 1,0,0,20
POINT LIGHT 1,0,0,0
SET LIGHT RANGE 1,100
SYNC
PRINT "Spot light created - press any key"
SYNC
WAIT KEY

REM *** Move point light down the y-axis ***
PRINT "Press key to move light"
SYNC
WAIT KEY
zpos = 20
DO
    POSITION LIGHT 1,0,0,zpos
    SYNC
    INC zpos
    WAIT 50
LOOP

REM *** End program ***
WAIT KEY
END
```

Activity 34.8

Type in and test the program given in LISTING-34.3 (*lights03.dbpro*).

Replace the sphere with a 20 by 20 plane. How does this affect the results?

Why has the spot light stopped working when we changed from a sphere to a plane?

When DarkBASIC Pro calculates lighting effects, it does so for each polygon that makes up a 3D shape. A single polygon can only have a single light shading. Planes are constructed from only two polygons - not nearly enough to create the subtle effects of a spot light. It only works on the sphere because it was constructed from a huge number of polygons.

The ROTATE LIGHT Statement

A spot or directional light can also be rotated to specific angles about all three axes using the ROTATE LIGHT statement whose format is given in FIG-34.17

FIG-34.17

The ROTATE LIGHT Statement



In the diagram:

<i>lightno</i>	is an integer value giving the ID of the light to be rotated.
<i>xangle</i>	is a real number giving the absolute angle to which the light is to be rotated about the x-axis.
<i>yangle</i>	is a real number giving the absolute angle to which the light is to be rotated about the y-axis.
<i>zangle</i>	is a real number giving the absolute angle to which the light is to be rotated about the z-axis.

Activity 34.9

Modify your previous program as follows:

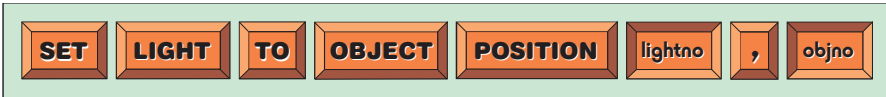
1. Remove the plane and return to a sphere as previously defined.
2. Set the spot to have an inner cone angle of 2 and outer cone angle of 8.
3. Position the spot light at (0,0,40).
4. In the DO..LOOP structure rotate the spot light about its y-axis rather than move it away from the sphere.

The SET LIGHT TO OBJECT POSITION Statement

A light can be moved to the position of some other object using the SET LIGHT TO OBJECT POSITION statement which has the format shown in FIG-34.18.

FIG-34.18

The SET LIGHT TO OBJECT POSITION Statement



In the diagram:

<i>lightno</i>	is an integer value giving the ID of the light to be moved.
<i>objno</i>	is an integer value giving the ID of the object to whose coordinates the light is to be moved. The light will be placed at the exact centre of the object specified.

For example, if we want light 1 to be moved to the position of object 10, then we would use the line:

```
SET LIGHT TO OBJECT POSITION 1,10
```

We could achieve the same effect using the rather more complex line:

```
POSITION LIGHT 1,OBJECT POSITION X(10),OBJECT POSITION Y(10),  
  ↵ OBJECT POSITION Z(10)
```

The program in LISTING-34.4 creates a row of spheres. A small cone with an accompanying point light then moves past the spheres. As the cone and its light move, each sphere, in turn, is affected by the light.

LISTING-34.4

Moving a Light with an
Object

```
REM *** Set up screen ***  
SET DISPLAY MODE 1280,1024,32  
COLOR BACKDROP RGB(20,20,20)  
BACKDROP ON  
  
REM *** Programmed screen updates ***  
SYNC ON  
  
REM *** Position camera ***  
AUTOCAM OFF  
POSITION CAMERA 0,0,-60  
POINT CAMERA 0,0,0  
  
REM *** Make and position sphere ***  
MAKE OBJECT SPHERE 2,10,100,100  
POSITION OBJECT 2,-36,0,0  
xpost = -24  
FOR objno = 3 TO 8  
  CLONE OBJECT objno,2  
  POSITION OBJECT objno,xpost,0,0  
  xpost = xpost + 12  
NEXT objno  
  
REM *** Make and position cone ***  
MAKE OBJECT CONE 10, 2  
POSITION OBJECT 10, -36,0,-10  
  
REM *** Hide default directional light ***  
HIDE LIGHT 0  
  
REM *** Create spot light ***  
MAKE LIGHT 1  
SET POINT LIGHT 1,-12,0,-10  
SET LIGHT RANGE 1,10  
  
DO  
  POSITION LIGHT 1,OBJECT POSITION X(10),OBJECT POSITION Y(10),  
  ↵ OBJECT POSITION Z(10)  
  MOVE OBJECT RIGHT 10,1  
  SYNC  
  WAIT 1  
LOOP  
  
REM *** End program ***  
END
```

Activity 34.10

Type in and test the program in LISTING-34.4 (*lights04.dbpro*). Change the first line in the DO loop to a SET LIGHT TO OBJECT POSITION statement.

The SET LIGHT TO OBJECT ORIENTATION Statement

A spot or directional light can be made to face in the same direction as some other object using the SET LIGHT TO OBJECT ORIENTATION statement which has the format shown in FIG-34.19.

FIG-34.19

The SET LIGHT
TO OBJECT
ORIENTATION
Statement



In the diagram:

lightno

is an integer value giving the ID of the light to be moved.

objno

is an integer value giving the ID of an object. The light specified will be made to point in the same direction as this object.

To make spot light 1 point in the same direction as object 10 we would use the statement:

```
SET LIGHT TO OBJECT ORIENTATION 1,10
```

The identical effect could be achieved using the line:

```
ROTATE LIGHT 1, OBJECT ANGLE X(10), OBJECT ANGLE Y(10),  
  ↳ OBJECT ANGLE Z(10)
```

The program in LISTING-34.5 is a variation on the previous program. This time a cube is positioned out from the centre sphere. The cube rotates about its own y-axis and a spot light rotates in synchronisation with the cube. A second light - a point light - has been added to make the cube itself more visible.

LISTING-34.5

Pointing a Light in the
Same Direction as an
Object

```
REM *** Set up screen ***  
SET DISPLAY MODE 1280,1024,32  
COLOR BACKDROP RGB(20,20,20)  
BACKDROP ON  
  
REM *** Use programmed screen updating ***  
SYNC ON  
  
REM *** Position camera ***  
AUTOCAM OFF  
POSITION CAMERA 0,0,-60  
POINT CAMERA 0,0,0  
  
REM *** Make and position spheres ***  
MAKE OBJECT SPHERE 2,10,100,100  
POSITION OBJECT 2,-36,0,0  
xpost = -24  
FOR objno = 3 TO 8  
  CLONE OBJECT objno,2  
  POSITION OBJECT objno,xpost,0,0  
  xpost = xpost + 12  
NEXT objno  
  
REM *** Make and position cube ***  
MAKE OBJECT CUBE 10, 2  
POSITION OBJECT 10, 0,0,-10
```

continued on next page

LISTING-34.5
(continued)

Pointing a Light in the Same Direction as an Object

```
REM *** Dull directed light by changing it to dark grey ***
COLOR LIGHT 0, RGB(50,50,50)

REM *** Create spot light at cube's position ***
MAKE LIGHT 1
SET SPOT LIGHT 1,10,30
POSITION LIGHT 1, 0,0,-10
POINT LIGHT 1,0,0,0
SET LIGHT RANGE 1,100

REM *** Illuminate cube with second light ***
MAKE LIGHT 2
SET POINT LIGHT 2, 2,4,-11
SET LIGHT RANGE 2, 10

REM *** Rotate cube and light about y-axis***
yrotate = 0
DO
    YROTATE OBJECT 10,yrotate
    ROTATE LIGHT 1,0,OBJECT ANGLE Y(10),0
    SET CURSOR 100,100
    PRINT OBJECT ANGLE Y(10)
    yrotate = WRAPVALUE(yrotate + 1)
    SYNC
    WAIT 1
LOOP

REM *** End program ***
END
```

Activity 34.11

Type in and test the program in LISTING-34.5 (*lights05.dbpro*).

Try using a SET LIGHT TO OBJECT ORIENTATION statement in place of the ROTATE LIGHT statement within the DO..LOOP structure.

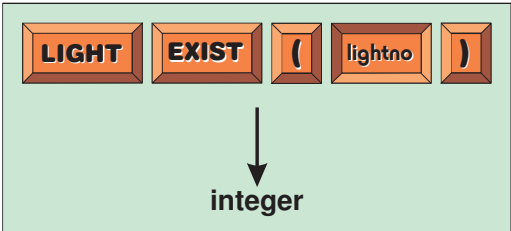
Retrieving Light Data

Various statements exist which allow us to retrieve information about lights. These are listed below.

The LIGHT EXIST Statement

The LIGHT EXIST statement returns 1 if a light of a given ID value exists; if the light does not exist, the statement returns zero. This statement has the format shown in FIG-34.20.

FIG-34.20
The LIGHT EXIST Statement



In the diagram:

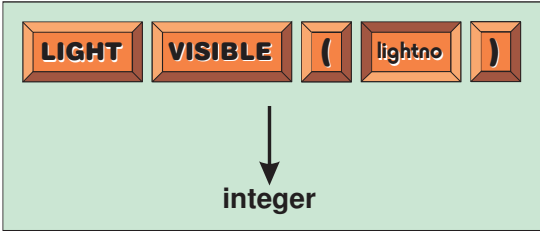
lightno is an integer value giving the ID of the light to be checked.

The LIGHT VISIBLE Statement

The LIGHT VISIBLE statement returns 1 if a specified light is currently switched on (that is, visible); otherwise zero is returned. This statement has the format shown in FIG-34.21.

FIG-34.21

The LIGHT VISIBLE Statement



In the diagram:

lightno

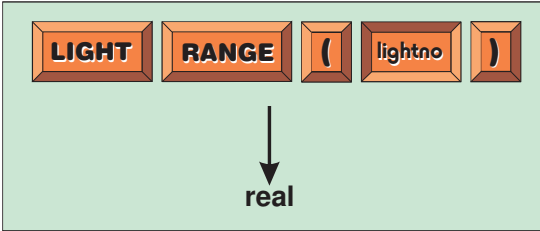
is an integer value giving the ID of the light to be checked.

The LIGHT RANGE Statement

The LIGHT RANGE statement returns a real number giving the range setting (as specified using the SET LIGHT RANGE statement) for a specified light. This statement has the format shown in FIG-34.22.

FIG-34.22

The LIGHT RANGE Statement



In the diagram:

lightno

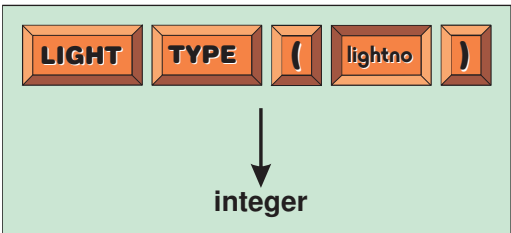
is an integer value giving the ID of the light to be checked.

The LIGHT TYPE Statement

The LIGHT TYPE statement returns an integer value giving a specified light's type. This statement has the format shown in FIG-34.23.

FIG-34.23

The LIGHT TYPE Statement



In the diagram:

lightno

is an integer value giving the ID of the light to be checked.

Possible values for the returned integer are:

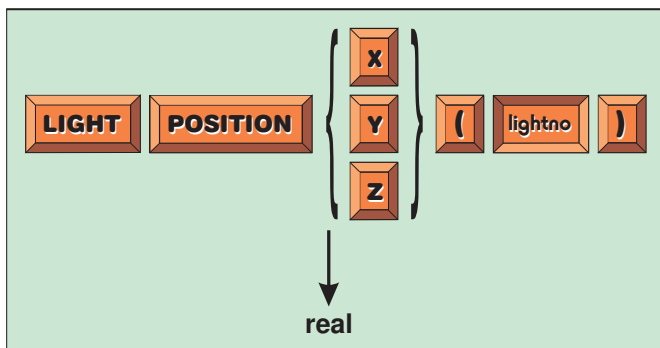
- 1 - directional light
- 2 - point light
- 3 - spot light

The LIGHT POSITION Statement

We can discover the position of a light (point or spot) using the LIGHT POSITION statement, each form of the statement returning a single ordinate (x, y or z). The statement has the format shown in FIG-34.24.

FIG-34.24

The LIGHT POSITION Statement



In the diagram:

X Y Z

Use the appropriate option to return the ordinate required.

lightno

is an integer value giving the ID of the light involved.

A real value is returned by this statement. We could record the exact position of light 1 using the following statements:

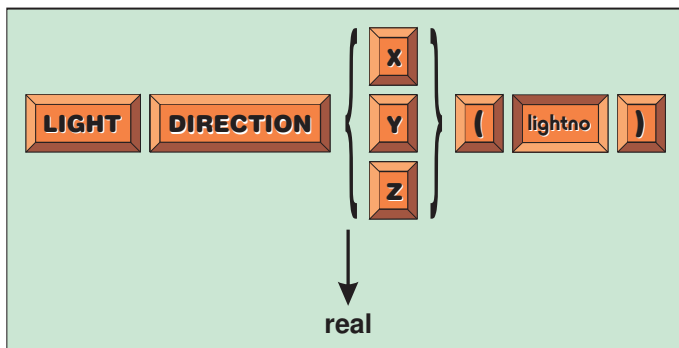
```
x# = LIGHT POSITION X(1)
y# = LIGHT POSITION Y(1)
z# = LIGHT POSITION Z(1)
```

The LIGHT DIRECTION Statement

The point at which a directional or spot light is aimed can be determined using the LIGHT DIRECTION statement. This statement, which can be used in any of three forms, returns the x, y, or z ordinate of the point at which the light is aimed. The statement has the format shown in FIG-34.25.

FIG-34.25

The LIGHT DIRECTION Statement



In the diagram:

X Y Z

Use the appropriate option to return the ordinate required.

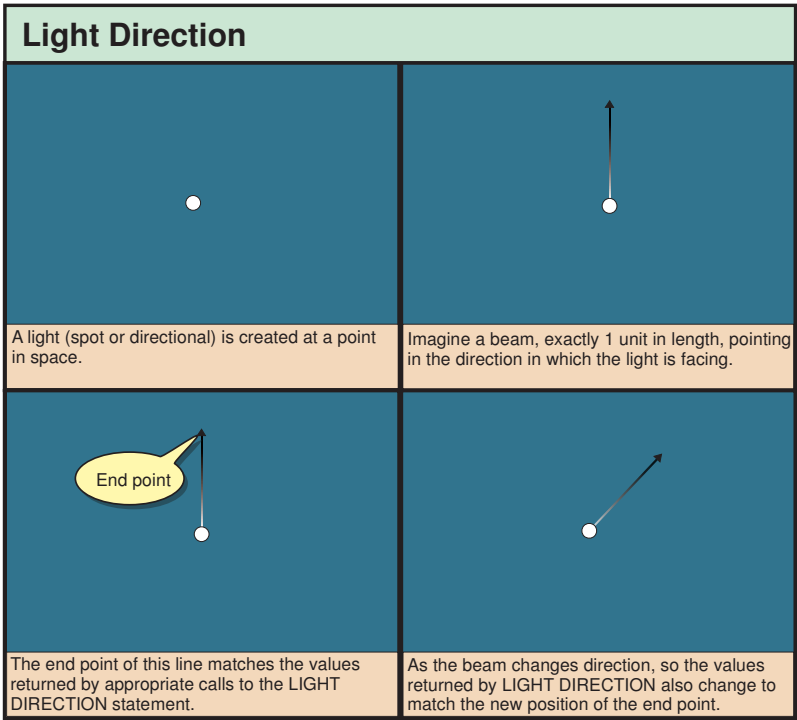
lightno

is an integer value giving the ID of the light involved.

A real value is returned by this statement. The value returned always lies between -1 and 1. FIG-34.26 shows how that value is calculated.

FIG-34.26

How the Value Returned by LIGHT DIRECTION is Calculated



Activity 34.12

Modify *lights05.dbpro* so that it displays the direction of the light in terms of the values returned by LIGHT DIRECTION rather than the angle of rotation.

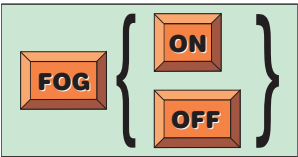
Fog

The FOG Statement

We can create a fog-like effect using FOG ON and disable the effect using FOG OFF. The FOG statement has the format shown in FIG-34.27.

FIG-34.27

The FOG Statement

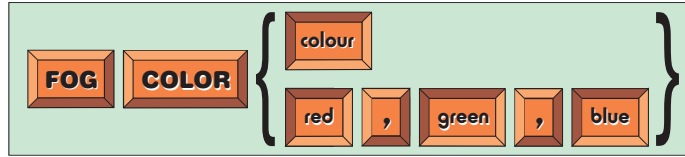


The FOG COLOR Statement

The fog's colour can be set using the FOG COLOR statement which has the format shown in FIG-34.28.

FIG-34.28

The FOG COLOR
Statement



In the diagram:

colour

is an integer value giving the colour of the fog.

red,green,blue

is a set of integer values giving the red, green and blue component values for the required fog colour.

The FOG DISTANCE Statement

The distance from the active camera at which the fog becomes so dense as to obscure any objects at that range can be set using the FOG DISTANCE statement which has the format shown in FIG-34.29.

FIG-34.29

The FOG DISTANCE
Statement



In the diagram:

distance

is an integer value giving the distance from the active camera at which objects become totally obscured by the fog.

The program in LISTING-34.6 demonstrates the use of the fog statements. The program creates an enclosing cube, inside which a line of spheres recede into the distance. The fog is switched on and the fog density varied.

LISTING-34.6

Creating Fog

```
REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP RGB(20,20,20)
BACKDROP ON

REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 20,0,60
POINT CAMERA 0,0,0

REM *** Switch off default directional light ***
HIDE LIGHT 0

REM *** Make and position spheres ***
MAKE OBJECT SPHERE 1,4,60,60
FOR objno = 2 to 8
  CLONE OBJECT objno,1
  POSITION OBJECT objno,0,0,objno*5
NEXT objno
```

continued on next page

LISTING-34.6

(continued)

Creating Fog

```
REM *** Create an enclosing cube ***
MAKE OBJECT CUBE 10,-200
POSITION OBJECT 10,0,-3,0

REM *** Create a light ***
MAKE LIGHT 1
POSITION LIGHT 1,10,1,40
SET LIGHT RANGE 1,50

REM *** Set up Fog ***
WAIT 1000
FOG ON
REM *** Colour fog ***
FOG COLOR RGB(60,10,90)

REM *** Vary fog density ***
FOR dist = 100 TO 20 STEP -1
    FOG DISTANCE dist
    WAIT 10
NEXT dist

REM *** End program ***
WAIT KEY
END
```

Activity 34.13

Type in and test the program in LISTING-34.6 (*lights06.dbpro*).

The SET OBJECT FOG Statement

Individual objects can be made to ignore fog effects (and hence appear totally unaffected by the fog) using the SET OBJECT FOG statement. This statement has the format shown in FIG-34.30.

FIG-34.30

The SET OBJECT FOG
Statement



In the diagram:

objno

is an integer value giving the ID of the object whose reaction to fog is to be changed.

fogflag

is 0 or 1. When set to zero, the object is unaffected by the fog; when set to 1, the object is affected by fog as normal.

Activity 34.14

In your previous program, create a moving cube using the following code:

```
WAIT 2000
MAKE OBJECT CUBE 11,1
POSITION OBJECT 11,5,0,0
POINT OBJECT 11,5,0,60
FOR move = 1 TO 60
    MOVE OBJECT 11,1
    WAIT 10
NEXT move
```

continued on next page

Activity 34.14 (continued)

The code should be inserted immediately before the final WAIT KEY statement.

Remove the FOR loop which varies the fog distance and replace it with a fixed fog distance of 50.

Run the modified program, observing the result.

Insert the line

```
SET OBJECT FOG 11,0
```

before the cube begins moving and check what difference this makes to the cube's appearance.

Lights and 3D Vectors

Again, there are a couple of statements that make use of 3D vectors. These statements are just alternatives to other light statements we've already covered, but are included here for completeness. 3D vectors themselves are covered in a later chapter.

The statements are summarised in TABLE-34.1.

TABLE-34.1

Lights and 3D Vectors
Statements

Statement	Parameters	Returned	Description
SET VECTOR3 TO LIGHT POSITION	vno, lightno	vno	The 3D vector with ID <i>vno</i> is set to hold the coordinates of <i>lightno</i> .
SET VECTOR3 TO LIGHT ROTATION	vno, lightno	vno	The 3D vector with ID <i>vno</i> is set to hold <i>lightno</i> 's rotation about each axis.

Summary

- There are four types of lighting in DarkBASIC Pro:
 - Ambient - rays from every possible direction; each surface lit equally.
 - Point - rays come from a single point in space and radiate in all directions from that point. The brightness grows weaker over distance.
 - Spot - rays come from a single point and radiate in one direction only. Rays form a cone shape with a bright inner cone and duller outer cone. Brightness grows weaker over distance.
 - Directional - light comes from an undefined distance point; all rays are parallel. Brightness does not diminish over distance.
- DarkBASIC Pro programs automatically create an ambient light and a directional light.

- The default ambient light has no ID value.
- The default directional light has ID zero.
- Use `HIDE LIGHT` to switch off a specific light.
- Use `SHOW LIGHT` to switch a light back on.
- Use `SET AMBIENT LIGHT` to set the brightness of the default ambient light.
- Use `COLOR AMBIENT` to set the colour of the ambient light.
- Use `MAKE LIGHT` to create a new light. The light created will be a point light.
- Use `DELETE LIGHT` to remove a light from RAM.
- Use `COLOR LIGHT` to set the colour of a light.
- Use `POSITION LIGHT` to place a light at a specific point in space (point and spot only).
- Use `SET LIGHT RANGE` to specify how far a light's brightness carries (point and spot only).
- Use `SET SPOT LIGHT` to change a light to a spot light.
- Use `SET DIRECTIONAL LIGHT` to change a light to a directional light.
- Use `SET POINT LIGHT` to change a light to a point light.
- Use `POINT LIGHT` to specify the direction of a light (spot and directional only).
- Use `ROTATE LIGHT` to rotate the direction of a light (spot and directional only).
- Use `SET LIGHT TO OBJECT POSITION` to move a light to the centre of a specified object.
- Use `SET LIGHT TO OBJECT ORIENTATION` to make a light point in the same direction as a specified object.
- Use `LIGHT EXIST` to check if a specified light exists.
- Use `LIGHT VISIBLE` to check if a specified light is switched on.
- Use `LIGHT RANGE` to check the range of a specified light (spot and point only).
- Use `LIGHT TYPE` to determine the type of a specified light.
- Use `LIGHT POSITION` to determine the position of a specified light (spot and point only).
- Use `LIGHT DIRECTION` to determine the end point of a 1 unit beam emitted from a specified light assumed to be at position (0,0,0) (spot and directional only).
- Spot and point lights only work effectively on surfaces constructed from many polygons.

- Use the FOG statement to switch a fog effect on or off.
- Use FOG COLOR to set the colour of the fog.
- Use FOG DISTANCE to set the distance at which fog becomes opaque.
- Use SET OBJECT FOG to determine how an object reacts to fog.
- The type of light to which a statement may be applied is shown below:

Statement	Light			
	A	D	P	S
COLOR AMBIENT LIGHT	✓			
COLOR LIGHT		✓	✓	✓
DELETE LIGHT		✓	✓	✓
HIDE LIGHT		✓	✓	✓
LIGHT DIRECTION		✓		✓
LIGHT EXIST		✓	✓	✓
LIGHT POSITION			✓	✓
LIGHT RANGE			✓	✓
LIGHT TYPE		✓	✓	✓
LIGHT VISIBLE		✓	✓	✓
MAKE LIGHT			✓	
POINT LIGHT		✓		✓
POSITION LIGHT			✓	✓
ROTATE LIGHT		✓		✓
SET AMBIENT LIGHT	✓			
SET DIRECTIONAL LIGHT			✓	✓
SET LIGHT RANGE			✓	✓
SET LIGHT TO OBJECT POSITION			✓	✓
SET LIGHT TO OBJECT ORIENTATION		✓		✓
SET POINT LIGHT		✓		✓
SET SPOT LIGHT		✓	✓	
SET VECTOR3 TO LIGHT POSITION			✓	✓
SET VECTOR3 TO LIGHT ROTATION		✓		✓
SHOW LIGHT		✓	✓	✓

Solutions

Activity 34.1

No solution required.

Activity 34.2

```
REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP RGB(0,200,0)
BACKDROP ON
REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA -10,10,20
POINT CAMERA 0,0,0
REM *** Make and position cube ***
MAKE OBJECT CUBE 2,5
POSITION OBJECT 2,0,2.5,0
REM *** Hide directional light ***
WAIT KEY
HIDE LIGHT 0
FOR amblight = 100 TO 0 STEP -1
  SET AMBIENT LIGHT amblight
  WAIT 50
NEXT amblight
REM *** End program ***
WAIT KEY
END
```

Activity 34.3

```
REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP RGB(0,200,0)
BACKDROP ON
REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA -10,10,20
POINT CAMERA 0,0,0
REM *** Make and position cube ***
MAKE OBJECT CUBE 2,5
POSITION OBJECT 2,0,2.5,0
REM *** Hide directional light ***
WAIT KEY
HIDE LIGHT 0
REM *** Make ambient blue ***
COLOR AMBIENT LIGHT RGB(0,0,255)
FOR amblight = 100 TO 0 STEP -1
  SET AMBIENT LIGHT amblight
  WAIT 50
NEXT amblight
REM *** End program ***
WAIT KEY
END
```

Activity 34.4

No solution required.

Activity 34.5

```
REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP RGB(20,20,20)
BACKDROP ON
REM *** Use programmed screen updating ***
SYNC ON
REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA -10,-3,20
POINT CAMERA 0,0,0
REM *** Make and position cubes ***
```

```
MAKE OBJECT CUBE 2,5
POSITION OBJECT 2,0,2.5,-10
MAKE OBJECT CUBE 3,5
POSITION OBJECT 3,0,10,0
SYNC:SYNC
REM *** Switch off directional light ***
WAIT KEY
HIDE LIGHT 0
SYNC
PRINT "Directional light off"
SYNC
REM *** Create new light ***
WAIT KEY
MAKE LIGHT 1
REM *** Make light yellow ***
COLOR LIGHT 1,255,255,0
SYNC
PRINT "New light created"
SYNC
REM *** End program ***
WAIT KEY
END
```

Activity 34.6

```
REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP RGB(20,20,20)
BACKDROP ON
REM *** Use programmed screen updating ***
SYNC ON
REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA -10,-3,20
POINT CAMERA 0,0,0
REM *** Make and position cubes ***
MAKE OBJECT CUBE 2,5
POSITION OBJECT 2,0,2.5,-10
MAKE OBJECT CUBE 3,5
POSITION OBJECT 3,0,10,0
SYNC:SYNC
REM *** Switch off directional light ***
WAIT KEY
HIDE LIGHT 0
SYNC
PRINT "Directional light off"
SYNC
REM *** Create new light ***
WAIT KEY
MAKE LIGHT 1
COLOR LIGHT 1,255,255,0
SYNC
PRINT "New light created"
SYNC
REM *** Move light away ***
y# = 0
DO
  POSITION LIGHT 1,0,y#,0
  y# = y# - 0.5
  SYNC
LOOP
REM *** End program ***
WAIT KEY
END
```

Activity 34.7

```
REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP RGB(20,20,20)
BACKDROP ON
REM *** Use programmed screen updating ***
SYNC ON
```

```

REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA -10,-3,20
POINT CAMERA 0,0,0
REM *** Make and position cubes ***
MAKE OBJECT CUBE 2,5
POSITION OBJECT 2,0,2.5,-10
MAKE OBJECT CUBE 3,5
POSITION OBJECT 3,0,10,0
SYNC:SYNC
REM *** Switch off directional light ***
WAIT KEY
HIDE LIGHT 0
SYNC
PRINT "Directional light off"
SYNC
REM *** Create new light ***
WAIT KEY
MAKE LIGHT 1
COLOR LIGHT 1,255,255,0
SYNC
PRINT "New light created"
SYNC
REM *** Set light range ***
SET LIGHT RANGE 1, 50
REM *** Move light away ***
y# = 0
DO
    POSITION LIGHT 1,0,y#,0
    y# = y# - 0.5
    SYNC
LOOP
REM *** End program ***
WAIT KEY
END

```

Activity 34.8

```

REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP RGB(20,20,20)
BACKDROP ON
REM *** Use programmed screen updating ***
SYNC ON
REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 0,0,20
POINT CAMERA 0,0,0
REM *** Make and position plane ***
MAKE OBJECT PLAIN 2,20,20
SYNC:SYNC
REM *** Switch off directional light ***
WAIT KEY
HIDE LIGHT 0
SYNC
PRINT "Directional light off - press any key"
SYNC
REM *** Create spot light ***
WAIT KEY
MAKE LIGHT 1
SET SPOT LIGHT 1,40,60
POSITION LIGHT 1,0,0,20
POINT LIGHT 1,0,0,0
SET LIGHT RANGE 1,100
SYNC
PRINT "Spot light created - press any key"
SYNC
WAIT KEY
REM *** Move point light down the y-axis ***
PRINT "Press key to move light"
SYNC
WAIT KEY
zpos = 20

```

```

DO
    POSITION LIGHT 1,0,0,zpos
    SYNC
    INC zpos
    WAIT 50
LOOP
REM *** End program ***
WAIT KEY
END

```

The spotlight effect appears to have stopped working.

Activity 34.9

```

REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP RGB(20,20,20)
BACKDROP ON
REM *** Use programmed screen updating ***
SYNC ON
REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 0,0,20
POINT CAMERA 0,0,0
REM *** Make and position sphere ***
MAKE OBJECT SPHERE 2,20,150,150
SYNC:SYNC
REM *** Switch off directional light ***
WAIT KEY
HIDE LIGHT 0
SYNC
PRINT "Directional light off - press any key"
SYNC
REM *** Create spot light ***
WAIT KEY
MAKE LIGHT 1
REM *** Set inner and outer cones ***
SET SPOT LIGHT 1,2,8
POSITION LIGHT 1,0,0,40
POINT LIGHT 1,0,0,0
SET LIGHT RANGE 1,100
SYNC
PRINT "Spot light created - press any key"
SYNC
WAIT KEY
REM *** Move point light down the y-axis ***
PRINT "Press key to move light"
SYNC
WAIT KEY
yangle# = 0
DO
    ROTATE LIGHT 1,0,yangle#,0
    SYNC
    yangle# = WRAPVALUE(yangle#+1)
    WAIT 10
LOOP
REM *** End program ***
WAIT KEY
END

```

You may have to adjust the increment to *yangle#* and the wait period depending on the power of the machine you are using.

Activity 34.10

```

REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP RGB(20,20,20)
BACKDROP ON
REM *** Programmed screen updates ***

```

```

SYNC ON
REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 0,0,-60
POINT CAMERA 0,0,0
REM *** Make and position sphere ***
MAKE OBJECT SPHERE 2,10,100,100
POSITION OBJECT 2,-36,0,0
xpost = -24
FOR objno = 3 TO 8
    CLONE OBJECT objno,2
    POSITION OBJECT objno,xpost,0,0
    xpost = xpost + 12
NEXT objno
REM *** Make and position cone ***
MAKE OBJECT CONE 10, 2
POSITION OBJECT 10, -36,0,-10
REM *** Hide default directional light ***
HIDE LIGHT 0
REM *** Create spot light ***
MAKE LIGHT 1
SET POINT LIGHT 1,-12,0,-10
SET LIGHT RANGE 1,10
DO
    SET LIGHT TO OBJECT POSITION 1,10
    MOVE OBJECT RIGHT 10,1
    SYNC
    WAIT 50
LOOP
REM *** End program ***
END

```

In the version of DarkBASIC Pro used (v6.0), the SET LIGHT TO OBJECT POSITION did not operate correctly, so you will have to revert to the original POSITION LIGHT statement.

Activity 34.11

```

REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP RGB(20,20,20)
BACKDROP ON
REM *** Use programmed screen updating ***
SYNC ON
REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 0,0,-60
POINT CAMERA 0,0,0
REM *** Make and position spheres ***
MAKE OBJECT SPHERE 2,10,100,100
POSITION OBJECT 2,-36,0,0
xpost = -24
FOR objno = 3 TO 8
    CLONE OBJECT objno,2
    POSITION OBJECT objno,xpost,0,0
    xpost = xpost + 12
NEXT objno
REM *** Make and position cube ***
MAKE OBJECT CUBE 10, 2
POSITION OBJECT 10, 0,0,-10
REM *** Dull directed light by changing it to dark grey ***
COLOR LIGHT 0, RGB(50,50,50)
REM *** Create spot light at cube's position ***
MAKE LIGHT 1
SET SPOT LIGHT 1,10,30
POSITION LIGHT 1, 0,0,-10
POINT LIGHT 1,0,0,0
SET LIGHT RANGE 1,100
REM *** Illuminate cube with new light ***
MAKE LIGHT 2

```

```

SET POINT LIGHT 2, 2,4,-11
SET LIGHT RANGE 2, 10
REM *** Rotate cube/light about y-axis***
yrotate = 0
DO
    YROTATE OBJECT 10,yrotate
    SET LIGHT TO OBJECT ORIENTATION 1,10
    SET CURSOR 100,100
    PRINT OBJECT ANGLE Y(10)
    yrotate = WRAPVALUE(yrotate + 1)
    SYNC
    WAIT 1
LOOP
REM *** End program ***
END

```

In DarkBASIC Pro (v6.0) the SET LIGHT TO OBJECT ORIENTATION does not achieve the required effect.

Activity 34.12

```

REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP RGB(20,20,20)
BACKDROP ON
REM *** Use programmed screen updating ***
SYNC ON
REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 0,0,-60
POINT CAMERA 0,0,0
REM *** Make and position spheres ***
MAKE OBJECT SPHERE 2,10,100,100
POSITION OBJECT 2,-36,0,0
xpost = -24
FOR objno = 3 TO 8
    CLONE OBJECT objno,2
    POSITION OBJECT objno,xpost,0,0
    xpost = xpost + 12
NEXT objno
REM *** Make and position cube ***
MAKE OBJECT CUBE 10, 2
POSITION OBJECT 10, 0,0,-10
REM *** Dull directed light by changing it to dark grey ***
COLOR LIGHT 0, RGB(50,50,50)
REM *** Create spot light at cube's position ***
MAKE LIGHT 1
SET SPOT LIGHT 1,10,30
POSITION LIGHT 1, 0,0,-10
POINT LIGHT 1,0,0,0
SET LIGHT RANGE 1,100
REM *** Illuminate cube with new light ***
MAKE LIGHT 2
SET POINT LIGHT 2, 2,4,-11
SET LIGHT RANGE 2, 10
REM *** Rotate cube/light about y-axis***
yrotate = 0
DO
    YROTATE OBJECT 10,yrotate
    ROTATE LIGHT 1,0,OBJECT ANGLE Y(10),0
    dirx# = LIGHT DIRECTION X(1)
    diry# = LIGHT DIRECTION Y(1)
    dirz# = LIGHT DIRECTION Z(1)
    SET CURSOR 100,100
    PRINT dirx#," ",diry#," ",dirz#
    yrotate = WRAPVALUE(yrotate + 1)
    SYNC
    WAIT 1
LOOP
REM *** End program ***
END

```

Activity 34.13

No solution required.

Activity 34.14

```
REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP RGB(20,20,20)
BACKDROP ON
REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 20,0,60
POINT CAMERA 0,0,0
REM *** Switch off directional light ***
HIDE LIGHT 0
REM *** Make and position spheres ***
MAKE OBJECT SPHERE 1,4,60,60
FOR objno = 2 TO 8
    CLONE OBJECT objno,1
    POSITION OBJECT objno,0,0,objno*5
NEXT objno
REM *** Create an enclosing cube ***
MAKE OBJECT CUBE 10,-200
POSITION OBJECT 10,0,-3,0
REM *** Create a light ***
MAKE LIGHT 1
POSITION LIGHT 1,10,1,40
SET LIGHT RANGE 1,50
REM *** Set up Fog ***
WAIT 1000
FOG ON
REM *** Colour fog ***
FOG COLOR RGB(60,10,90)
REM *** Set fog distance to 50 ***
FOG DISTANCE 50
REM *** fly in cube ***
WAIT 2000
MAKE OBJECT CUBE 11,1
POSITION OBJECT 11,5,0,0
POINT OBJECT 11,5,0,60
FOR move = 1 TO 60
    MOVE OBJECT 11,1
    WAIT 50
NEXT move
REM *** End program ***
WAIT KEY
END
```

The cube is seen emerging from the fog.

When the line

```
SET OBJECT FOG 11,0
```

is added before the final FOR loop, the cube is unaffected by the fog.

Accessing Limb Data

Adding a Mesh to an Object

Changing a Limb's Mesh

Constructing Models from a Root Object and Limbs

Creating an Opening Door

Creating and Using Meshes

Creating an Object from a Mesh

Glueing an Object to a Limb

Loading and Saving Meshes

Manipulating Limbs

Offsetting and Rotating Limbs

Saving a Model in DBO Format

Scaling Limbs

Texturing and Colouring Limbs

Introduction

A 3D object's details (its vertices, texture, etc.) can be held by the computer in many forms. In DarkBASIC Pro, the commonest of these is the **object** format, but other options are available. One format holds an object in the form of a **mesh**.

To differentiate between the two formats we will refer to them as **objects** and **meshes**. The most obvious difference between a mesh and an object is that meshes are not visible on screen and do not retain texture details!

However, a mesh can be saved to, or loaded from, a file.

Meshes in themselves are of limited use. Their main purpose is as building blocks for more complex shapes, as we'll see later in this chapter.

Handling Meshes

The MAKE MESH FROM OBJECT Statement

To create a mesh we need to convert an existing object to mesh format. This is done using the MAKE MESH FROM OBJECT statement which has the format shown in FIG-35.1.

FIG-35.1

The MAKE MESH
FROM OBJECT
Statement



In the diagram:

meshno

is an integer value specifying the ID to be assigned to the mesh being created. No other mesh in the program may be assigned the same value.

objno

is an integer value giving the ID of the existing 3D object whose details are to be used to create the mesh.

The 3D object is unaffected by this operation.

The program in LISTING-35.1 creates a sphere and uses this to create a mesh; the sphere is then deleted, leaving us with a mesh, but nothing visible on the screen.

LISTING-35.1

Creating a Mesh

```
REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP 0
BACKDROP ON

REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 0,10,-100
POINT CAMERA 0,0,0
```

continued on next page

LISTING-35.1
(continued)

Creating a Mesh

```
REM *** Make sphere ***
MAKE OBJECT SPHERE 1,10
WAIT KEY

REM *** Make mesh from sphere ***
MAKE MESH FROM OBJECT 1,1

REM *** Delete the original object ***
DELETE OBJECT 1

REM *** End program ***
WAIT KEY
END
```

Activity 35.1

Type in and test the program given in LISTING-35.1 (*limbs01.dbpro*).

The SAVE MESH Statement

Once created, a mesh can be saved to a file using the SAVE MESH statement. The format for this statement is given in FIG-35.2.

FIG-35.2

The SAVE MESH
Statement



In the diagram:

- filename* is a string specifying the name to be given to the file in which the mesh data is to be saved. The string may include path information.
- meshno* is an integer value giving the ID of the mesh to be saved.

Files are saved in DirectX format and, therefore, should be given a .x extension when being named.

For example, to save mesh 1 to a file named *sphere01.x* in a folder named *meshdata* on the C: drive, we would use the line:

```
SAVE MESH "C:\meshdata\sphere01.x",1
```

If the file already exists, its existing data will be overwritten; if the file does not exist, it will be created. Any folders named in the path details must already exist or the save operation will fail.

If path details are omitted, then the file will be saved in the current folder - normally, this will be the folder containing the program code.

Activity 35.2

Modify your last program so that the sphere created is saved in a file called *sphere01.x* in the current project's folder.

Modify the program to texture the sphere using *marble.bmp* before converting it to a mesh. Save the resulting mesh in file *sphere02.x*.

The LOAD MESH Statement

A previously saved mesh (or most other 3D objects saved in .x format) can be loaded into a program using the LOAD MESH statement which has the format given in FIG-35.3.

FIG-35.3

The LOAD MESH Statement



In the diagram:

filename

is a string specifying the name of the file containing the mesh.
The string may include path information.

meshno

is an integer value specifying the ID to be assigned to the mesh when loaded.

For example, we could load the file *sphere01.x* (as created by the last program) into a mesh with ID 1 using the line:

```
LOAD MESH "C:\limbs01\sphere01.x", 1
```

The MAKE OBJECT Statement

If we are to make a loaded mesh visible, then we need to turn it back into a regular object. This can be achieved using the MAKE OBJECT statement which has the format shown in FIG-35.4.

FIG-35.4

The MAKE OBJECT Statement



In the diagram:

objno

is an integer value specifying the ID to be assigned to the object being created.

meshno

is an integer value specifying the ID of the mesh to be used to create the object.

0

Use this option (zero) if no texture is to be applied to the new object.

imageno

is an integer value specifying the ID of the image to be used to texture the object. This image must have been loaded previously.

In the program in LISTING-35.2, the previously saved mesh, *sphere01.x*, is loaded and converted to an object. A marble texture is applied to the new object.

LISTING-35.2

Creating an Object from a Mesh

This code assumes that the file *sphere01.x* is in the same folder as the program code.

```
REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP 0
BACKDROP ON
REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 0,10,-100
POINT CAMERA 0,0,0
REM *** Load the mesh ***
LOAD MESH "sphere01.x",1
REM *** Convert mesh to object ***
MAKE OBJECT 1,1,1
REM *** End program ***
WAIT KEY
END
```

Activity 35.3

Type in and test the program given in LISTING-35.2 (*limbs02.dbpro*).
Make sure that the files *sphere01.x*, *sphere02.x* and *marble.bmp* have been copied into the current project's folder.
Modify the program to load *sphere02.x* instead of *sphere01.x*. Is the object created from loading *sphere02.x* textured?

The DELETE MESH Statement

A mesh can be erased from RAM using the DELETE MESH statement which has the format shown in FIG-35.5.

FIG-35.5

The LOAD MESH Statement



In the diagram:

meshno is an integer value specifying the ID of the mesh to be deleted.

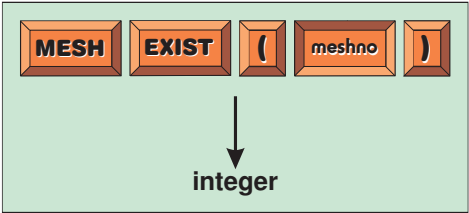
After a mesh has been used to create an object, we could use this command to delete the original mesh.

The MESH EXIST Statement

We can check to see if a mesh of a specific ID currently exists using the MESH EXIST statement which has the format shown in FIG-35.6.

FIG-35.6

The MESH EXIST Statement



In the diagram:

meshno is an integer value specifying the mesh ID that is to be checked.

If a mesh of the specified number exists, then 1 is returned by the statement, otherwise zero is returned.

Summary

- A mesh is a 3D shape held in a different format from the more usual 3D object.
- Mesh shapes are not normally visible on the screen.
- Use MAKE MESH FROM OBJECT to create a mesh.
- Use SAVE MESH to save a mesh to a file.
- Use LOAD MESH to load a mesh file.
- Use MAKE OBJECT to create an object from an existing mesh.
- When creating an object from a mesh, a texture can be applied to the object once it is created.
- Use DELETE MESH to delete an existing mesh from RAM.
- Use MESH EXIST to check if a mesh of a specified number currently exists.

Introduction

A regular object and one or more meshes can be linked together to create a **group object** or **model**. This can be useful, allowing us to do something as simple as applying different textures to different parts of the group object, or to make the moving of one object result in the automatic movement of a second object. When a mesh is linked to a standard 3D object, the mesh is known as a **limb** and the original object is known as the **root**.

Getting Started

At the core of any model is the basic root object. This can be created in the normal way. We'll start by creating a sphere object:

```
MAKE OBJECT SPHERE 1,10
```

Now we can create a second object and convert it to a mesh, deleting the object when we're done:

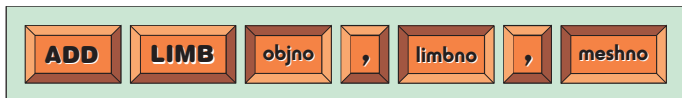
```
REM *** Make a box ...***  
MAKE OBJECT BOX 2,10,10,0.1  
REM *** ...and turn it into a mesh ***  
MAKE MESH FROM OBJECT 1,2  
REM *** Now delete the original box object ***  
DELETE OBJECT 2
```

The ADD LIMB Statement

Using the ADD LIMB statement we can attach our mesh to the sphere object. Since many meshes may be attached to a model, it is necessary to assign an attachment (limb) number to each mesh. The format for the ADD LIMB statement is given in FIG-35.7.

FIG-35.7

The ADD LIMB Statement



In the diagram:

<i>objno</i>	is an integer value identifying the object to which the mesh is being joined.
<i>limbno</i>	is an integer value specifying the limb number to be assigned to the mesh when it is joined to the object. Each limb joined to the object must have a unique limb number.
<i>meshno</i>	is an integer value identifying the mesh from which the limb is to be created.

For example, we could join our box mesh to the sphere object using the line

```
ADD LIMB 1,1,1
```

The program in LISTING-35.3 joins a box mesh to a sphere object and then allows the user to move the camera to view the resulting model.

LISTING-35.3

Adding a Limb to an Object

```
REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP RGB(0,200,0)
BACKDROP ON

REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 0,10,-100
POINT CAMERA 0,0,0

REM *** Make and position sphere ***
MAKE OBJECT SPHERE 1,10
MAKE OBJECT BOX 2,10,10,0.1

REM *** Make mesh from object ***
MAKE MESH FROM OBJECT 1,2

REM *** Delete original object ***
DELETE OBJECT 2

REM *** Attach mesh to object
ADD LIMB 1,1,1

REM *** Allow user to move camera ***
DO
    CONTROL CAMERA USING ARROWKEYS 0,1,1
LOOP

REM *** End program ***
END
```

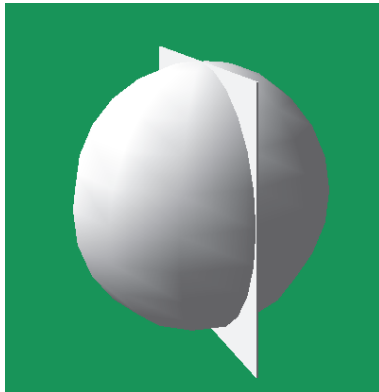
Activity 35.4

Type in and test the program given in LISTING-35.3 (*limbs03.dbpro*).

The output of this program is shown in FIG-35.8.

FIG-35.8

Adding a Limb to a Sphere



Notice that the limb's centre is positioned at the centre of the object to which it is joined.

Once we have created a group object by adding a limb to a standard object, all the items within that group become linked. As a result, if the root object is moved, its limbs will also move.

Activity 35.5

In your last program add the lines

```
REM *** Move object ***  
WAIT KEY  
POSITION OBJECT 1, -10, 0, 0
```

immediately after the mesh is attached to the sphere.

Run the program and check that the sphere's limb moves along with the sphere.

In fact, any other operation which is applied to the root object, affects all the items in the group object. For example, using the line `HIDE OBJECT 1` in the last program, would hide not only the sphere, but also the limb attached to it. If we colour or texture the object, its limbs will also take on the same colour or texture.

Activity 35.6

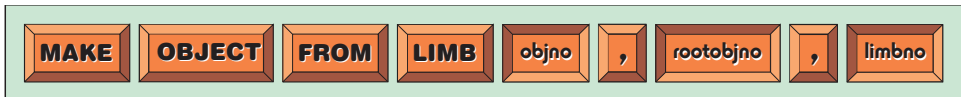
Using the `WAIT KEY` statement to initiate each action, modify your last program so that the root object changes colour to red and then is textured using the image *marble.bmp*.

The MAKE OBJECT FROM LIMB statement

It's possible to turn a limb into an object using the `MAKE OBJECT FROM LIMB` statement which has the format shown in FIG-35.9.

FIG-35.9

The `MAKE OBJECT FROM LIMB`
Statement



In the diagram:

objno is an integer value specifying the ID to be assigned to the object that is to be created.

rootobjno is an integer value identifying the limb's root object.

limbno is an integer value giving the number of the limb being used to create an object.

For example, in the last program we could create a new object, 5, from limb number 1 of group object 1 using the line:

```
MAKE OBJECT FROM LIMB 5, 1, 1
```

The new object will not retain any colour or texture details from the root object.

Activity 35.7

In your last program, create object 2 from limb 1 of the group object. Move the newly created object to position (20,0,0).

Once you've tested the program, return the program to its original coding.

The OFFSET LIMB Statement

When a limb is first added, its centre will be placed at the root object's centre. However, the limb can be repositioned using the OFFSET LIMB statement. This statement has the format given in FIG-35.10.

FIG-35.10

The OFFSET LIMB Statement



In the diagram:

objno is an integer value giving the limb's root object.

limbno is an integer value specifying the limb to be moved. A limb of this value must previously have been attached to the root object.

x,y,z are real values giving the offset distances of the limb along each axis, as measured from its original position.

For example, we could move the box attached to the sphere to the back of that sphere (a distance of 5 units along the z-axis) using the line:

```
OFFSET LIMB 1,1,0,0,5
```

Activity 35.8

Add the above line to your last program immediately after the limb is attached to the sphere. How does this change the position of the limb?

Now we'll add a second box limb at the front of the sphere using the lines:

```
ADD LIMB 1,2,1  
OFFSET LIMB 1,2,0,0,-5
```

Notice that the mesh used to create the first limb is also used to create the second limb.

Activity 35.9

Add the above lines to your program and check out the position of the two limbs.

The ROTATE LIMB Statement

In positioning a limb, it can be rotated about any of the three axes using the ROTATE

LIMB statement which has the format shown in FIG-35.11.

FIG-35.11 The ROTATE LIMB Statement



In the diagram:

<i>objno</i>	is an integer value identifying the limb's root object.
<i>limbno</i>	is an integer value specifying the limb to be rotated.
<i>xang</i>	is a real number giving the angle of rotation about the x-axis.
<i>yang</i>	is a real number giving the angle of rotation about the y-axis.
<i>zang</i>	is a real number giving the angle of rotation about the z-axis.

All angles are given in degrees.

The next lines create a third box limb on the sphere and place that limb on the base of the sphere:

```
ADD LIMB 1,3,1
ROTATE LIMB 1,3,90,0,0
OFFSET LIMB 1,3,0,-5,0
```

As a result of adding this code, we get the model shown in FIG-35.12.

FIG-35.12

A Rotated Limb is Added
to the Sphere



Activity 35.10

Modify your last program so that a total of 6 limbs are added to the sphere creating a cube around the sphere.

Remove the code which moves the model and allows the camera to be controlled by the user.

Replace it with code which makes the model rotate about all three of its axes.

The SCALE LIMB Statement

An individual limb within a model can be resized using the SCALE LIMB statement which has the format shown in FIG-35.13.

FIG-35.13 The SCALE LIMB Statement



In the diagram:

objno is the integer value specifying the limb's root object.

limbno is an integer value identifying the limb to be scaled.

xperc is a real number giving the new size of the object's x dimension as a percentage of its original size in that dimension. For example, a value of 100.0 will retain the original size, while 200.0 would double the object's length in the x dimension, and 50.0 would halve it.

yperc is a real number giving the new size of the object's y dimension as a percentage of its original size in that dimension.

zperc is a real number giving the new size of the object's z dimension as a percentage of its original size in that dimension.

Activity 35.11

In your previous program, modify the width (x) and height (y) of limb 4 to be 200% of their original size; do not modify the depth of the limb.

Modify the program again to make limb 4 grow slowly in width and height from 100% to 200% and then back again. This action should happen continuously.

Remove all changes made in this Activity before continuing.

We have ended up with a cube! But surely it would have been much easier to use the MAKE OBJECT CUBE statement. However, the advantage of using limbs is that we can now colour or texture the limbs individually.

The COLOR LIMB Statement

To colour an individual limb, we use the COLOR LIMB statement which has the format shown in FIG-35.14.

FIG-35.14

The COLOR LIMB Statement



In the diagram:

- objno* is an integer value identifying the limb's root object.
- limbno* is an integer value specifying the limb to be coloured.
- colour* is an integer value giving the colour to be assigned to the limb. Often this will be the value returned by an RGB statement.

For example, limb 1 of object 1 could be assigned the colour yellow using the line

```
COLOR LIMB 1,1,RGB(255,255,0)
```

Activity 35.12

Modify your last program so that the cube's sides are coloured as follows:

Side	Colour
1	yellow
2	red
3	blue
4	magenta
5	cyan
6	green

The TEXTURE LIMB Statement

Rather than just colour a limb we can texture it with an image by using the TEXTURE LIMB statement whose format is shown in FIG-35.15.

FIG-35.15

The TEXTURE LIMB Statement



In the diagram:

- objno* is an integer value identifying the limb's root object.
- limbno* is an integer value specifying the limb to be textured.
- imgno* is an integer value giving the ID of the image to be used as the texture.

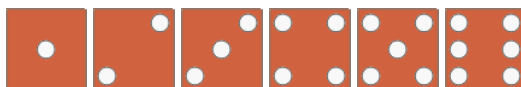
For example, limb 1 of object 1 could be textured using the image *onespot.bmp* using the lines:

```
LOAD IMAGE "onespot.bmp", 1
TEXTURE LIMB 1,1,1
```

By using a set of six images, we can create the dice shown in FIG-35.16.

FIG-35.16

Creating a Dice



Images Used



Final Group Object

The program which produces the dice is given in LISTING-35.4.

LISTING-35.4

Texturing a Dice

```
REM *** Main program ***
SetUpScreen()
CreateDiceShape()
TextureDice()
RotateDice()
REM *** End program ***
END

FUNCTION SetUpScreen()
    REM *** Set up screen ***
    SET DISPLAY MODE 1280,1024,32
    COLOR BACKDROP RGB(0,200,0)
    BACKDROP ON
    REM *** Position camera ***
    AUTOCAM OFF
    POSITION CAMERA 0,20,-100
    POINT CAMERA 0,0,0
ENDFUNCTION

FUNCTION CreateDiceShape()
    REM *** Make and position sphere ***
    MAKE OBJECT SPHERE 1,10
    MAKE OBJECT BOX 2,10,10,0.1
    REM *** Make mesh from object ***
    MAKE MESH FROM OBJECT 1,2
    REM *** Delete original object ***
    DELETE OBJECT 2
    REM *** Attach back ***
    ADD LIMB 1,1,1
    OFFSET LIMB 1,1,0,0,5
    REM *** Add front ***
    ADD LIMB 1,2,1
    OFFSET LIMB 1,2,0,0,-5
    REM *** Add bottom ***
    ADD LIMB 1,3,1
    ROTATE LIMB 1,3,90,0,0
    OFFSET LIMB 1,3,0,-5,0
    REM *** Add top ***
    ADD LIMB 1,4,1
    ROTATE LIMB 1,4,90,0,0
    OFFSET LIMB 1,4,0,5,0
    REM *** Add left side ***
    ADD LIMB 1,5,1
    ROTATE LIMB 1,5,0,90,0
    OFFSET LIMB 1,5,-5,0,0
    REM *** Add right side ***
    ADD LIMB 1,6,1
```

continued on next page

LISTING-35.4
(continued)

Texturing a Dice

```
ROTATE LIMB 1,6,0,90,0
OFFSET LIMB 1,6,5,0,0
ENDFUNCTION

FUNCTION TextureDice()
    REM *** Load Images ***
    LOAD IMAGE "Spot1.bmp",1
    LOAD IMAGE "Spot2.bmp",2
    LOAD IMAGE "Spot3.bmp",3
    LOAD IMAGE "Spot4.bmp",4
    LOAD IMAGE "Spot5.bmp",5
    LOAD IMAGE "Spot6.bmp",6
    REM *** Colour top red ***
    TEXTURE LIMB 1,1,1
    TEXTURE LIMB 1,2,6
    TEXTURE LIMB 1,3,2
    TEXTURE LIMB 1,4,5
    TEXTURE LIMB 1,5,3
    TEXTURE LIMB 1,6,4
ENDFUNCTION

FUNCTION RotateDice()
    DO
        PITCH OBJECT UP 1, 1
        TURN OBJECT LEFT 1,1
        ROLL OBJECT LEFT 1,1
        WAIT 10
    LOOP
ENDFUNCTION
```

Activity 35.13

Type in and test the program given in LISTING-35.4 (*limbs04.dbpro*).

Activity 35.14

Create a new project (*limbsvideo.dbpro*). Create a cube using a sphere and six box limbs. Colour all limbs blue. Load the video *vd.mpg* on limb 2 of the cube and make the cube rotate about all three of its axes.

The SCALE LIMB TEXTURE Statement

Just as we can scale the texture applied to an object, so we can also scale the texture of a limb. This is done using the SCALE LIMB TEXTURE which has the format shown in FIG-35.17.

FIG-35.17

The SCALE LIMB TEXTURE Statement



In the diagram:

objno is an integer value specifying the limb's root object.

limbno is an integer value identifying the limb whose texture is to be scaled.

Uscale is a real number specifying the multiplication

factor along the U axis. Values less than 1 will result in only part of the image being used. Values greater than 1 will result in duplication of the image over the limb's surface.

Vscale

is a real number specifying the multiplication factor along the V axis. Values less than 1 will result in only part of the image being used. Values greater than 1 will result in duplication of the image over the limb's surface.

For example, we could make the single spot on limb 1 of our dice look like four spots by duplicating the texture twice along both axes. This would require the statement:

```
SCALE LIMB TEXTURE 1,1,2.0,2.0
```

Activity 35.15

Modify your *limbs04.dpro* program's *RotateDice()* function so that the dice is stationary with the single spot facing the camera.

Add a SCALE LIMB TEXTURE statement to the main section so that the single spot looks like six spots.

When the program is working as required, remove the SCALE LIMB TEXTURE statement and resave your program.

Activity 35.16

Write a program (*Act3516.dpro*) to create a corridor as shown in the image below. Each surface of the corridor should be a limb of a zero-sized root object. The functions within the program should be:

Files Used:

Floor - floor_m_01.bmp
Ceiling - ceil_m_03.bmp
Wall - wall_m_44.bmp

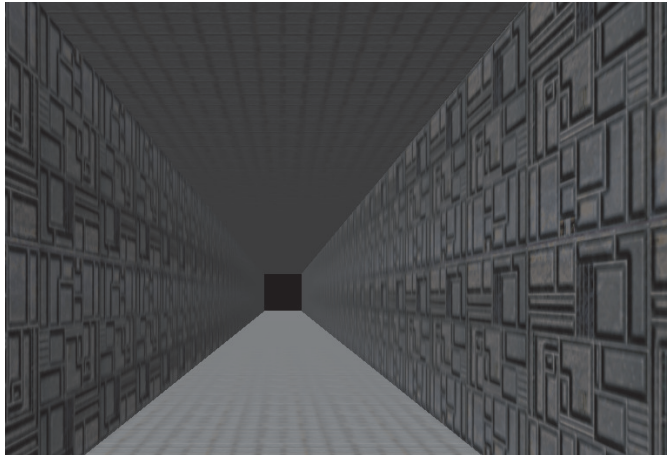
SetUpScreen

Sets the display mode and positions the camera at (0,4,-100)

CreateModel

Constructs the corridor

The main section should allow the user to move the camera using the arrow keys.



The SCROLL LIMB TEXTURE Statement

The texture image can be mapped onto a limb with a varying degree of offset along either the U or V axes. The overall effect is to modify which part of the texture image is placed at the top left corner of the limb. This is achieved using the SCROLL LIMB TEXTURE statement which has the format shown in FIG-35.18.

FIG-35.18 The SCROLL LIMB TEXTURE Statement



In the diagram:

<i>objno</i>	is an integer value specifying the limb's root object.
<i>limbno</i>	is an integer value identifying the limb whose texture is to be offset.
<i>Uoffset</i>	is a real number specifying the offset along the x-axis. This value should lie between 0 and 1.
<i>Voffset</i>	is a real number specifying the offset along the y-axis. This value should lie between 0 and 1.

The result of using the statement

`SCROLL LIMB 1,1,0.5,0.5`

on the one spot side of the dice is shown in FIG-35.19.

FIG-35.19

Scrolling a Limb's Texture

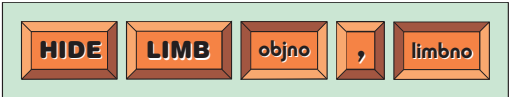


The HIDE LIMB Statement

Individual limbs within a group object can be hidden using the HIDE LIMB statement which has the format shown in FIG-35.20.

FIG-35.20

The HIDE LIMB Statement



In the diagram:

<i>objno</i>	is an integer value specifying the limb's root object.
<i>limbno</i>	is an integer value identifying the limb to be hidden.

Activity 35.17

Modify *limbs04.dbpro* to hide limb 1 of the dice.

Notice that the root object (the sphere) becomes visible when we remove the side panel. However, we can avoid this by giving the root object a diameter of zero. This won't affect the size or visibility of the limbs attached to the sphere.

Activity 35.18

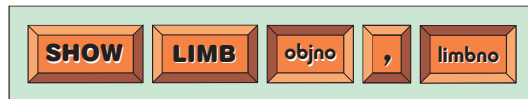
Modify your last program so that the sphere is created with a diameter of zero.

The SHOW LIMB Statement

A hidden limb can be made visible again using the SHOW LIMB statement which has the format shown in FIG-35.21.

FIG-35.21

The SHOW LIMB
Statement



In the diagram:

objno

is an integer value specifying the limb's root object.

limbno

is an integer value identifying the limb to be redisplayed.

Activity 35.19

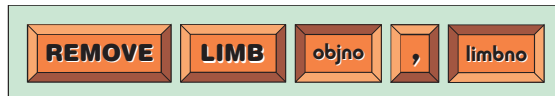
In *limbs04.dbpro*, modify the main section so that limb 1 of the dice reappears after a key press.

The REMOVE LIMB Statement

A limb can be deleted using the REMOVE LIMB statement which has the format shown in FIG-35.22.

FIG-35.22

The REMOVE LIMB
Statement



In the diagram:

objno

is an integer value specifying the limb's root object.

limbno

is an integer value identifying the limb to be deleted.

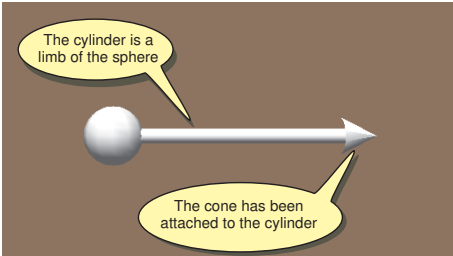
The LINK LIMB Statement

Although every limb in a group model must be added to the root object, a limb need not be attached directly to that object. Limbs can also be attached (or linked) to

existing limbs within the model. For example, in FIG-35.23 we see a model with a sphere as its root object to which a cylinder has been attached. Linked to the cylinder is a cone.

FIG-35.23

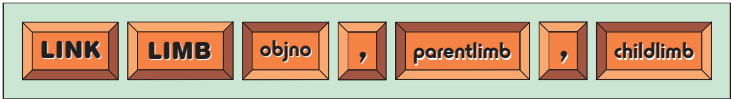
Linking a New Limb to an Existing Limb



The cone limb is known as the **child** limb, while the limb to which it is joined (the cylinder) is known as the **parent** limb. This effect is achieved using the LINK LIMB statement which has the format shown in FIG-35.24.

FIG-35.24

The LINK LIMB Statement



In the diagram:

- objno* is an integer value specifying the limb's root object.
- parentlimb* is an integer value identifying to which limb the child limb is to be attached.
- childlimb* is an integer value giving the ID of the limb being attached to the parent limb.

The steps involved in adding a child limb are shown in FIG-35.25.

FIG-35.25

Creating a Child Limb

Start by creating the root object: MAKE OBJECT SPHERE 1,8	Create and scale the objects to be used as limbs: MAKE OBJECT CYLINDER 2,10 SCALE OBJECT 2, 20,300,20 MAKE OBJECT CONE 3,5	Convert objects to meshes: MAKE MESH FROM OBJECT 1,2 MAKE MESH FROM OBJECT 2,3 and delete the original objects: DELETE OBJECT 2 DELETE OBJECT 3
Add the meshes as limbs of the sphere: ADD LIMB 1,1,1 ADD LIMB 1,2,2	Link the cone as a child of the cylinder: LINK LIMB 1,1,2	Move the cone to the end of the cylinder: OFFSET LIMB 1,2,0,15,0

It is best to attach and position child limbs before moving the parent limb from its original position.

Once attached to a parent limb, a child limb will move, rotate or resize automatically when the parent limb is altered. The program in LISTING-35.5 demonstrates the use of a child limb, and shows how that child limb is affected when the parent limb is moved.

LISTING-35.5

Using Child Limbs

```
REM *** Main program ***
SetUpScreen()
CreateModel()
ManipulateModel()
REM *** End program ***
WAIT KEY
END

FUNCTION SetUpScreen()
    REM *** Set up screen ***
    SET DISPLAY MODE 1280,1024,32
    COLOR BACKDROP RGB(120,67,39)
    BACKDROP ON
    REM *** Set up camera ***
    AUTOCAM OFF
    POSITION CAMERA 0,0,-100
    POINT CAMERA 0,0,0
ENDFUNCTION

FUNCTION CreateModel()
    REM *** Make model root ***
    MAKE OBJECT SPHERE 1,8
    REM *** Create objects used as limbs ***
    MAKE OBJECT CYLINDER 2,10
    SCALE OBJECT 2, 20,300,20
    MAKE OBJECT CONE 3,5
    REM *** Convert limb objects to meshes ***
    MAKE MESH FROM OBJECT 1,2
    MAKE MESH FROM OBJECT 2,3
    REM *** Delete limb objects ***
    DELETE OBJECT 2
    DELETE OBJECT 3
    REM *** Add limbs to root ***
    ADD LIMB 1,1,1
    ADD LIMB 1,2,2
    REM *** Link cone as child of cylinder ***
    LINK LIMB 1,1,2
    REM *** Move cone to end of cylinder ***
    OFFSET LIMB 1,2,0,15,0
ENDFUNCTION

FUNCTION ManipulateModel()
    REM *** Move cylinder up ***
    WAIT KEY
    FOR y = 1 TO 15
        OFFSET LIMB 1,1,0,y,0
        WAIT 100
    NEXT y
    REM *** Rotate whole model ***
    WAIT KEY
    FOR angle = 0 TO 45
        ROLL OBJECT LEFT 1,1
        WAIT 100
    NEXT angle
ENDFUNCTION
```

Activity 35.20

Type in and test the program in LISTING-35.5 (*limbs05.dbpro*).

Find out what happens to the cone when the cylinder is deleted, then return the program to its previous state.

The CHANGE MESH Statement

The mesh used on a specific limb of a model can be changed using the CHANGE MESH statement. This allows us to change the shape of a limb. The CHANGE MESH statement has the format shown in FIG-35.26.

FIG-35.26

The CHANGE MESH Statement



In the diagram:

objno

is an integer value specifying the limb's root object.

limbno

is an integer value identifying the limb whose mesh is to be changed.

meshno

is an integer value giving the ID of the new mesh to be assigned to the limb.

To change the cone in the model created in the last program to a sphere, we would use the lines:

MAKE OBJECT SPHERE 2,5	`Create sphere
MAKE MESH FROM OBJECT 3,2	`Change it into a mesh
CHANGE MESH 1,2,3	`Replace cone with sphere

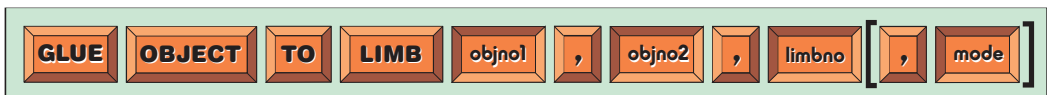
Activity 35.21

Modify the main section of your last program to change the cone to a sphere. Activate the change with a key press from the user.

The GLUE OBJECT TO LIMB Statement

An object can be temporarily linked to a limb. Such an object would then move in unison with that limb (even though it may not be visually attached to the limb). This can be useful to create the effect of a player picking up an item or to show items within a moving container. To attach an object to a limb, we use the statement GLUE OBJECT TO LIMB which has the format shown in FIG-35.27.

FIG-35.27 The GLUE OBJECT TO LIMB Statement



In the diagram:

objno1

is an integer value specifying the object to be glued to the limb.

objno2

is an integer value specifying the limb's root object.

limbno

is an integer value identifying the limb to which *objno1* is to be glued.

mode

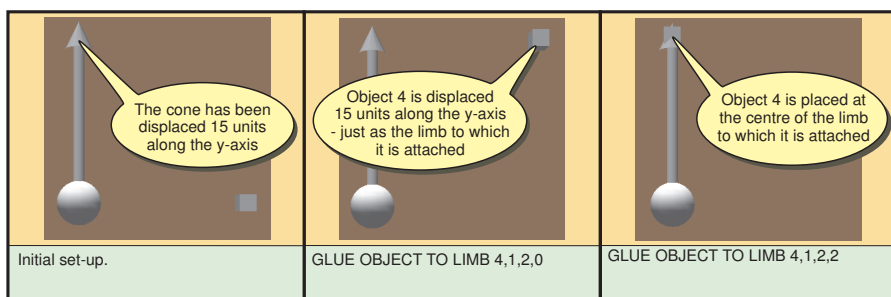
0, 1, or 2. This determines how the glued object attaches to the limb.

- 0 - any displacement that has been applied to the limb is also applied to the object. This is the default behaviour if the *mode* value is omitted from the statement.
- 1 - appears to result in a similar effect to the zero option.
- 2 - the object will be placed at the centre of the limb to which it has been glued.

FIG-35.28 shows the effect of the *mode* parameter when a cube is glued to the cone of the model we created earlier.

FIG-35.28

The Effects of the *mode* Parameter of the GLUE OBJECT TO LIMB Statement



We'll see the `GLUE OBJECT TO LIMB` statement in action by adding a new routine, *DemoGlue()*, to the last program. This new routine creates a cube object, moves the cylinder limb (and hence the cone limb) over to the cube before attaching the cube to the cone. After a key press, the whole model is rotated back to its starting position, with the cube following the movement. The new function is coded as:

```
FUNCTION DemoGlue()  
  REM *** Create and position cube ***  
  MAKE OBJECT CUBE 4,3  
  POSITION OBJECT 4,30,0,0  
  
  REM *** Rotate model so that cone is beside cube ***  
  FOR angle = 0 TO 85  
    ROLL OBJECT RIGHT 1,1  
    WAIT 50  
  NEXT angle  
  REM *** Glue cube to cone  
  GLUE OBJECT TO LIMB 4,1,2,0  
  WAIT KEY  
  
  REM *** Rotate model back to start position ***  
  FOR angle = 0 TO 85  
    ROLL OBJECT LEFT 1,1  
    WAIT 50  
  NEXT angle  
ENDFUNCTION
```

Activity 35.22

Make sure *limbs05.dbpro* matches the original listing.

Add the following lines - which move the cylinder - to the end of the *CreateModel()* function:

```
REM *** Position cylinder ***  
OFFSET LIMB 1,1,0,15,0
```

Add the code for *DemoGlue()* to your program.

Delete the function *ManipulateModel()*.

Change the line in the main section of the program from

```
ManipulateModel()  
to  
DemoGlue()
```

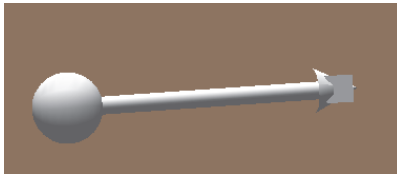
Run the new version of the program. What happens when the cube is linked to the cone?

Modify the GLUE OBJECT TO LIMB statement to use mode 1 and then mode 2. How does each change affect the result?

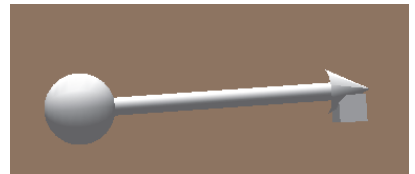
We can use POSITION OBJECT to reposition the glued object relative to the limb to which it is joined - but we need to take into account the rotation of the cube's axes. For example, rather than have the cube placed within the centre of the cone (as you observed in the last Activity) we could make the cube appear to be "glued" to the cone's surface by moving the cube slightly (see FIG-35.29).

FIG-35.29

Moving the Object after
Glueing



The Cube Attached to the Centre of the Cone



The Cube Repositioned Slightly after Attachment

From the moment an object is glued to a limb, the object's new position becomes its new origin (0,0,0) and the axes are orientated with those of the limb to which it is attached. For our cube, the consequences are shown in FIG-35.30.

So to move the cube down slightly (as shown in FIG-35.29) we need to use the line:

```
POSITION OBJECT 4,2,0,0
```

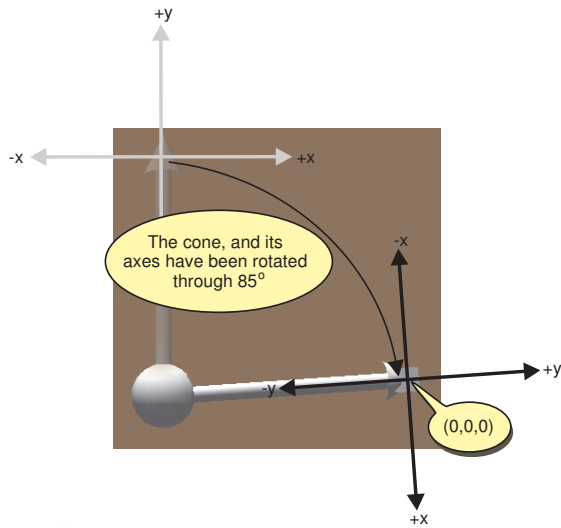
Notice that the movement is along the x-axis.

Activity 35.23

Try adding the POSITION OBJECT statement to your previous program immediately after the GLUE statement.

FIG-35.30

How an Object's Axes and Spatial Origin move when Glued to a Limb

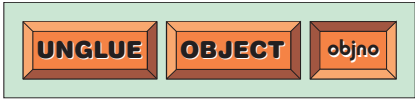


The UNGLUE OBJECT Statement

To detach an object from a group model, we need to use the UNGLUE OBJECT statement which has the format shown in FIG-35.31.

FIG-35.31

The UNGLUE OBJECT Statement



In the diagram:

objno is an integer value specifying the object to be unglued.

In our previous program we could unglue the cube from the cone using the line:

```
UNGLUE OBJECT 4
```

Activity 35.24

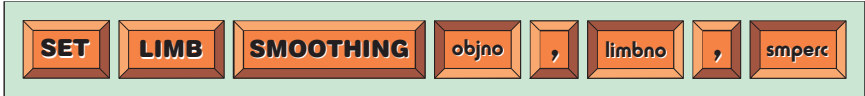
Modify function *DemoGlue()* in your last program so that the cylinder swings back down to 85°, unglues the cube, and then swings back up to the vertical position. Use a key press to activate each operation. How is the cube affected when it is unglued from the cone?

The SET LIMB SMOOTHING Statement

When two polygons drawn in different planes meet, often a sharp edge is created. By using the SET LIMB SMOOTHING statement, this affect can be smoothed out where it occurs within a specific limb. The SET LIMB SMOOTHING statement has the format shown in FIG-35.32.

FIG-35.32

The SET LIMB SMOOTHING Statement



In the diagram:

objno is an integer value specifying limb's root object.

limbno

is an integer value identifying the limb whose edges are to be smoothed.

smperc

is a integer percentage giving the degree of smoothing required. A value of 0 gives no smoothing effect while 100 gives maximum smoothing.

Activity 35.25

In your last program:

Comment out the call to *DemoGlue()*.

In *SetUpScreen()*, change the camera's position to (0,0,-55).

In the main section add code to smooth the cone limb after a key press (set the *smperc* value to 100).

Observe the subtle effect of the smoothing operation on the cone.

Creating Doors

Most doors are hinged on one side and open by rotating about the hinged edge. When we need to create a door in a game, it is simple enough to create a thin box with the appropriate texture (see FIG-35.33), but it takes a bit more effort to get the door to swing open. Using ROTATE OBJECT won't work because the door will rotate about its centre rather than its edge.

FIG-35.33

Creating a Door



To create a door that will swing open we need to resort to making the box a limb. We can then open the door by rotating the root object. The necessary code is shown in LISTING-35.6.

LISTING-35.6

Opening a Hinged Door

```
SetupScreen()  
CreateDoor()  
WAIT KEY  
SwingDoor(80)  
WAIT KEY  
END  
  
FUNCTION SetUpScreen()  
  REM *** Set screen resolution and position camera ***  
  SET DISPLAY MODE 1280, 1024,32  
  COLOR BACKDROP 0
```

LISTING-35.6

(continued)

Opening a Hinged Door

```
BACKDROP ON
AUTOCAM OFF
POSITION CAMERA 0,4,-10
POINT CAMERA 0,0,0
ENDFUNCTION

FUNCTION CreateDoor()
    #CONSTANT doorimg      1
    #CONSTANT doorhingeobj 1
    #CONSTANT doorobj      2
    #CONSTANT doormesh     1
    #CONSTANT doorlimb     1

    REM *** Load texture image ***
    LOAD IMAGE "door_05.jpg",doorimg
    REM *** Create root object ***
    MAKE OBJECT SPHERE doorhingeobj,0
    REM *** Add door ***
    MAKE OBJECT BOX doorobj,3,5,0.1
    MAKE MESH FROM OBJECT doormesh,doorobj
    DELETE OBJECT doorobj
    ADD LIMB doorhingeobj,doorlimb,doormesh
    TEXTURE LIMB doorhingeobj,doorlimb,doorimg
    OFFSET LIMB doorhingeobj,doorlimb,1.5,0,0
ENDFUNCTION

FUNCTION SwingDoor(angle)
    REM *** Swing door 1 degree at a time ***
    FOR c = 1 TO angle
        ROTATE OBJECT doorhingeobj,0,c,0
        WAIT 50
    NEXT c
ENDFUNCTION
```

Activity 35.26

Type in and test the program in LISTING-35.6 (*limbs06.dbpro*).

What happens if the door swings past the 90° mark?

Modify the *SwingDoor()* routine so that the door swings from its current starting place and allow the delay between each 1° move to be specified as a parameter.

Retrieving Limb Data

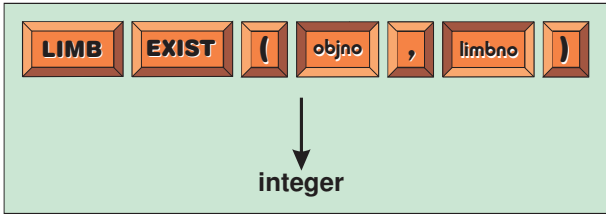
There are several statements available to determine various limb settings. These statements are described below.

The LIMB EXIST Statement

We can check to see if a limb of a specific number exists within a model using the LIMB EXIST statement which has the format shown in FIG-35.34.

FIG-35.34

The LIMB EXIST
Statement



In the diagram:

<i>objno</i>	is an integer value specifying the limb's root object.
<i>limbno</i>	is an integer value giving the limb number to be checked.

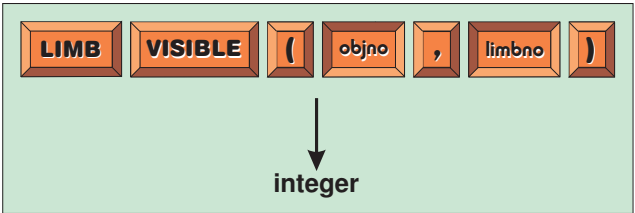
If the specified limb exists, 1 is returned, otherwise zero is returned.

The LIMB VISIBLE Statement

To check if a limb is visible, we can use the LIMB VISIBLE statement which has the format shown in FIG-35.35.

FIG-35.35

The LIMB VISIBLE Statement



In the diagram:

<i>objno</i>	is an integer value specifying the limb's root object.
<i>limbno</i>	is an integer value giving the limb number to be checked.

If the specified limb is visible, 1 is returned, otherwise zero is returned.

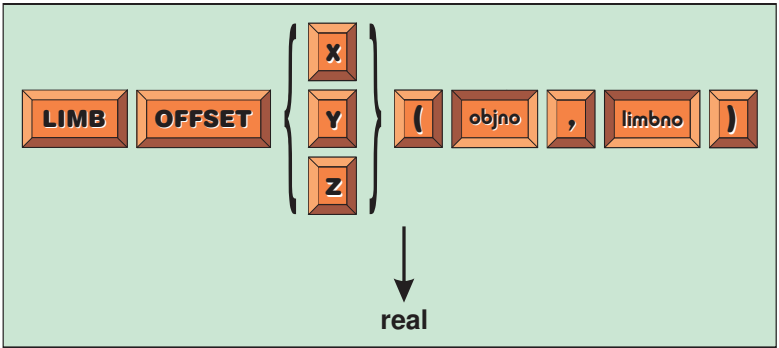
The LIMB OFFSET Statement

When a limb is first added to the root object (or another limb), the new limb is positioned at the centre of its parent component. In this position the new limb has an offset of zero in all three directions (x, y, and z). However, if we now move the new limb (as we moved the cone to the end of the cylinder in LISTING-35.5) an offset has been applied to that limb.

The current offset values of a limb can be determined using the LIMB OFFSET statement which has the format shown in FIG-35.36.

FIG-35.36

The LIMB OFFSET Statement



In the diagram:

X,Y,Z

Use one of these options to retrieve the offset value along the required axis.

objno

is an integer value identifying the root object to which the limb in question is attached.

limbno

is an integer value giving the number of the limb whose offset value is to be discovered.

If a limb is attached to the root object, then the values returned by this statement relate to the limb centre's displacement from the root object's centre. If a limb is a child limb then the values returned are measured from the parent limb's centre to the child limb's centre.

Activity 35.27

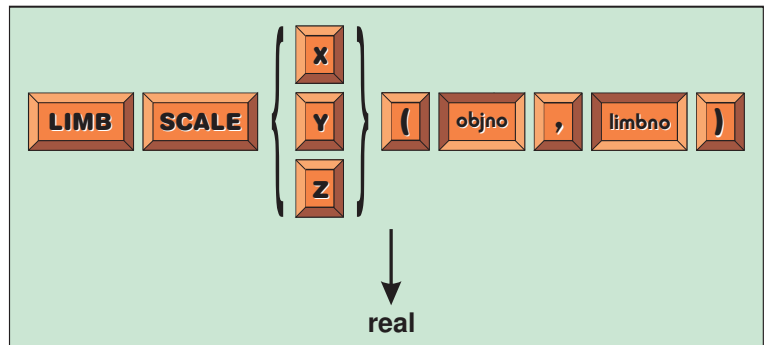
Modify your last version of *limbs05.dpro* to display the offset values in each direction of both the cylinder and the cone.

The LIMB SCALE Statement

If a limb is resized after being attached to a group object, the degree to which a limb has been scaled in each dimension can be determined using the LIMB SCALE statement, which has the format shown in FIG-35.37.

FIG-35.37

The LIMB SCALE Statement



In the diagram:

X,Y,Z

Use one of these options to retrieve the scale value along the required axis.

objno

is an integer value identifying the root object to which the limb in question is attached.

limbno

is an integer value giving the number of the limb whose scale value is to be returned.

For example, we could determine the scaling that has been applied to the cylinder (limb 1) in our previous program using the lines:

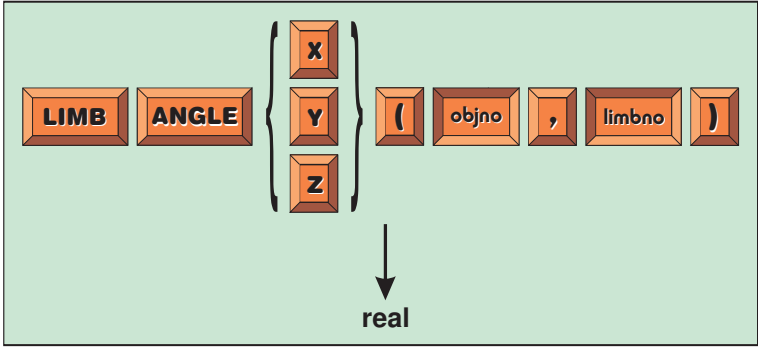
```
xscale# = LIMB SCALE X(1,1)
yscale# = LIMB SCALE Y(1,1)
zscale# = LIMB SCALE Z(1,1)
```

The LIMB ANGLE Statement

If a limb is rotated independently of its root or parent component (using ROTATE LIMB), the degree to which a limb has been rotated about each of its local axes can be determined using the LIMB ANGLE statement which has the format shown in FIG-35.38.

FIG-35.38

The LIMB ANGLE Statement



In the diagram:

- X,Y,Z

Use one of these options to retrieve the angle of rotation about the required axis.
- objno

is an integer value identifying the root object to which the limb in question is attached.
- limbno

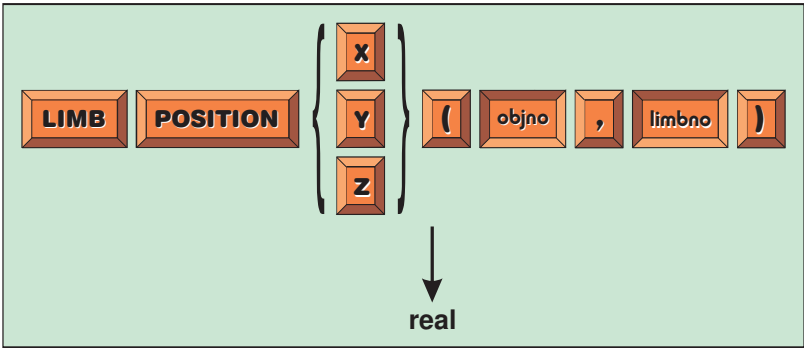
is an integer value giving the number of the limb whose angle of rotation is to be returned.

The LIMB POSITION Statement

The position of the centre of a limb within the 3D world can be discovered using the LIMB POSITION statement which has the format shown in FIG-35.39.

FIG-35.39

The LIMB POSITION Statement



In the diagram:

- X,Y,Z

Use one of these options to retrieve the position of the limb in the required direction.
- objno

is an integer value identifying the root object to which the limb in question is attached.
- limbno

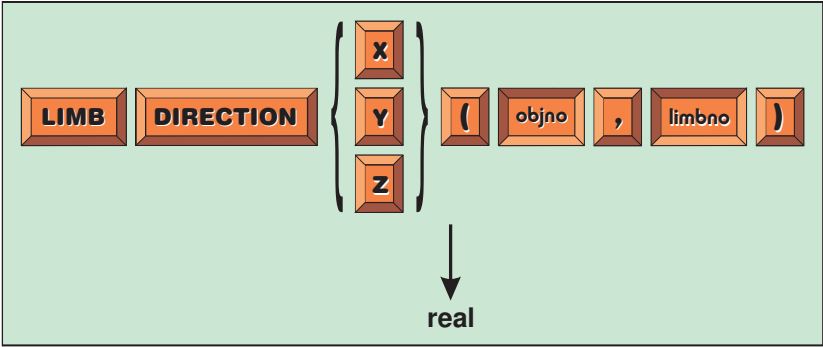
is an integer value giving the number of the limb whose position is to be returned.

The LIMB DIRECTION Statement

Whereas LIMB ANGLE returns the angle of a limb relative to its own local axes, the LIMB DIRECTION gives the angle of the limb relative to the main world axes. The statement has the format shown in FIG-35.40.

FIG-35.40

The LIMB DIRECTION Statement



In the diagram:

- X,Y,Z

Use one of these options to retrieve the direction of the limb along the required axis.
- objno

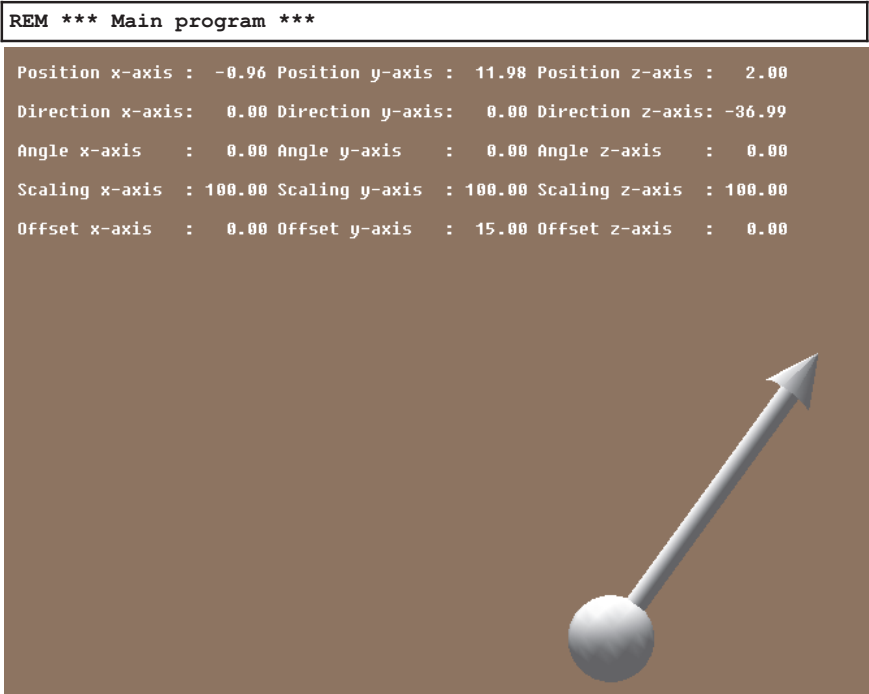
is an integer value identifying the root object to which the limb in question is attached.
- limbno

is an integer value giving the number of the limb whose world angle of rotation is to be returned.

FIG-35.41 shows a screen dump from the program in LISTING-35.7 which displays all the limb details for the cone in our previous model as it carries out various actions.

FIG-35.41

Displaying Limb Details



LISTING-35.7

Displaying Limb Data

```
SetUpScreen()
CreateModel()
REPEAT
    HandleUser()
UNTIL key$="5"
END

FUNCTION HandleUser()
    REM *** Display options ***
    SET CURSOR 0,100
    PRINT "1 - Rotate arm"
    PRINT "2 - Rotate cone"
    PRINT "3 - Move model"
    PRINT "4 - Scale cone"
    PRINT "5 - QUIT"
    SYNC
    REM *** Get user's choice ***
    REPEAT
        key$ = INKEY$()
    UNTIL key$ >= "1" AND key$ <= "5"
    REM *** Execute option chosen ***
    SELECT key$
        CASE "1"
            RotateArm()
            WAIT KEY
        ENDCASE
        CASE "2"
            RotateCone()
            WAIT KEY
        ENDCASE
        CASE "3"
            POSITION OBJECT 1,OBJECT POSITION X(1)-5, 0,0
            WAIT KEY
        ENDCASE
        CASE "4"
            ScaleCone()
            WAIT KEY
        ENDCASE
    ENDSELECT
ENDFUNCTION

FUNCTION SetUpScreen()
    REM *** Set up screen ***
    SET DISPLAY MODE 1280,1024,32
    COLOR BACKDROP RGB(120,67,39)
    BACKDROP ON
    AUTOCAM OFF
    POSITION CAMERA 0,0,-100
    POINT CAMERA 0,0,0
    SYNC ON
ENDFUNCTION

FUNCTION CreateModel()
    REM *** Make model root ***
    MAKE OBJECT SPHERE 1,8
    REM *** Create objects used as limbs ***
    MAKE OBJECT CYLINDER 2,10
    SCALE OBJECT 2, 20,300,20
    MAKE OBJECT CONE 3,5
    REM *** Convert limb objects to meshes ***
    MAKE MESH FROM OBJECT 1,2
    MAKE MESH FROM OBJECT 2,3
    REM *** Delete limb objects ***
    DELETE OBJECT 2
    DELETE OBJECT 3
```

continued on next page

REM *** Add limbs to root ***

LISTING-35.7

(continued)

Displaying Limb Data

```
ADD LIMB 1,1,1
ADD LIMB 1,2,2
REM *** Link cone as child of cylinder ***
LINK LIMB 1,1,2
OFFSET LIMB 1,2,0,15,0
OFFSET LIMB 1,1,0,15,0
SYNC
ENDFUNCTION

FUNCTION RotateCone()
  REM *** Rotate cone 1 degree at a time ***
  FOR angle = 1 TO 90
    ROTATE LIMB 1,2,0,0,angle
    REM *** Show limb data ***
    DisplayLimbData(1,2)
    SYNC
    WAIT 50
  NEXT angle
ENDFUNCTION

FUNCTION ScaleCone()
  REM *** Scale the cone 1% at a time ***
  FOR scale = 100 TO 50 STEP -1
    SCALE LIMB 1,2,100,scale,100
    REM *** Display limb data ***
    DisplayLimbData(1,2)
    SYNC
    WAIT 50
  NEXT scale
ENDFUNCTION

FUNCTION RotateArm()
  REM *** Rotate model showing limb data ***
  FOR angle = 0 TO 90
    ROLL OBJECT RIGHT 1,1
    DisplayLimbData(1,2)
    SYNC
    WAIT 50
  NEXT angle
  WAIT KEY
  REM *** Rotate model back to start position ***
  FOR angle = 0 TO 90
    ROLL OBJECT LEFT 1,1
    DisplayLimbData(1,2)
    SYNC
    WAIT 50
  NEXT angle
ENDFUNCTION

FUNCTION DisplayLimbData(objno,limbno)
  REM *** Get limb data ***
  xoffset# = LIMB OFFSET X(objno,limbno)
  yoffset# = LIMB OFFSET Y(objno,limbno)
  zoffset# = LIMB OFFSET Z(objno,limbno)
  xangle# = LIMB ANGLE X(objno,limbno)
  yangle# = LIMB ANGLE Y(objno,limbno)
  zangle# = LIMB ANGLE Z(objno,limbno)
  xpost# = LIMB POSITION X(objno,limbno)
  ypost# = LIMB POSITION Y(objno,limbno)
  zpost# = LIMB POSITION Z(objno,limbno)
  xdir# = LIMB DIRECTION X(objno,limbno)
  ydir# = LIMB DIRECTION Y(objno,limbno)
  zdir# = LIMB DIRECTION Z(objno,limbno)
  xscale# = LIMB SCALE X(objno,limbno)
  yscale# = LIMB SCALE Y(objno,limbno)
```

continued on next page

zscale# = LIMB SCALE Z(objno,limbno)

LISTING-35.7

(continued)

Displaying Limb Data

```
REM *** Display data on screen ***
SET CURSOR 100,70
PRINT "Position x-axis : ", Format$(xpost#,6,2),
  " Position y-axis : ",Format$(ypost#,6,2),
  " Position z-axis : ",Format$(zpost#,6,2)
SET CURSOR 100,100
PRINT "Direction x-axis: ", Format$(xdir#,6,2),
  " Direction y-axis: ",Format$(ydir#,6,2),
  " Direction z-axis: ",Format$(zdir#,6,2)
SET CURSOR 100,130
PRINT "Angle x-axis      : ", Format$(xangle#,6,2),
  " Angle y-axis       : ",Format$(yangle#,6,2),
  " Angle z-axis       : ",Format$(zangle#,6,2)
SET CURSOR 100,160
PRINT "Scaling x-axis   : ", Format$(xscale#,6,2),
  " Scaling y-axis    : ",Format$(yscale#,6,2),
  " Scaling z-axis    : ",Format$(zscale#,6,2)
SET CURSOR 100,190
PRINT "Offset x-axis   : ", Format$(xoffset#,6,2),
  " Offset y-axis    : ",Format$(yoffset#,6,2),
  " Offset z-axis    : " ,Format$(zoffset#,6,2)
ENDFUNCTION
```

The *Format\$()* routine is described in the May 2006 Support page of our website.

```
FUNCTION Format$(v#,totaldigits,decplaces)
  REM *** Check number's sign ***
  IF v# < 0
    sign = -1
  ELSE
    sign = 1
  ENDIF
  REM *** Round number ***
  v# = sign*ABS(v#+0.5/10^decplaces)
  REM *** Convert number to string ***
  v$ = STR$(v#)
  REM *** Find position of decimal point ***
  FOR size = 1 TO LEN(v$)
    IF MID$(v$,size) = "."
      EXIT
    ENDIF
  NEXT size
  REM *** Extract required decimal places from string ***
  result$ = LEFT$(v$,size+decplaces)
  REM *** Pad string to size required ***
  WHILE LEN(result$) < totaldigits
    result$ = " "+result$
  ENDWHILE
ENDFUNCTION result$
```

The purpose of each routine is described briefly below:

SetUpScreen	Initialises the screen resolution and camera position.
HandleUser	Displays user options, accepts keyboard selection, and executes the option chosen.
CreateModel	Creates the 3D model used in the program.
RotateCone	Rotates the cone relative to the parent cylinder limb.
ScaleCone	Scales the cone after it is attached to the cylinder.

RotateArm	Rotates the complete model 90° about the z-axis.
DisplayLimbData	Displays all details about the specified limb.
Format\$	Displays a real number to a specified format.

Activity 35.28

Type in and test the program in LISTING-35.7 (*limbs07.dbpro*).

The PERFORM CHECKLIST FOR OBJECT LIMBS Statement

It is possible to create a checklist giving details of the structure of a group object. This is done using the PERFORM CHECKLIST FOR OBJECT LIMBS statement which takes the format shown in FIG-35.42.

FIG-35.42

The PERFORM
CHECKLIST FOR
OBJECT LIMBS
Statement



In the diagram:

objno is an integer value giving the ID of the root object whose limbs are to be listed.

For example, we could create a checklist of the limbs for a model whose root object had the ID 5 using the line:

```
PERFORM CHECKLIST FOR OBJECT LIMBS 5
```

The checklist produced contains details of the root object and all limbs added to that object.

To access the checklist created by this command we need to use the following statements:

These statements were
covered in pages 364 -
366.

```
CHECKLIST QUANTITY
CHECKLIST STRING$
```

The program in LISTING-35.8 demonstrates the use of the PERFORM CHECKLIST FOR OBJECT LIMBS statement, displaying the details of the model containing the sphere, cylinder and cone that we created earlier.

LISTING-35.8

Listing Limb Names

```
REM *** Main section ***
SetupScreen()
CreateModel()
ModelDetails()
WAIT KEY
END

FUNCTION SetUpScreen()
    SET DISPLAY MODE 1280,1024,32
    AUTOCAM OFF
    POSITION CAMERA 0,0,-100
    POINT CAMERA 0,0,0
    SYNC ON
ENDFUNCTION
```

continued on next page

LISTING-35.8
(continued)

Listing Limb Names

```
FUNCTION CreateModel()  
    REM *** Make model root ***  
    MAKE OBJECT SPHERE 1,8  
    REM *** Create objects used as limbs ***  
    MAKE OBJECT CYLINDER 2,10  
    SCALE OBJECT 2,20,300,20  
    MAKE OBJECT CONE 3,5  
    REM *** Convert limb objects to meshes ***  
    MAKE MESH FROM OBJECT 1,2  
    MAKE MESH FROM OBJECT 2,3  
    REM *** Delete limb objects ***  
    DELETE OBJECT 2  
    DELETE OBJECT 3  
    REM *** Add limbs to root ***  
    ADD LIMB 1,1,1  
    ADD LIMB 1,2,2  
    REM *** Link cone as child of cylinder ***  
    LINK LIMB 1,1,2  
    REM *** Move cone to end of cylinder ***  
    OFFSET LIMB 1,2,0,15,0  
    OFFSET LIMB 1,1,0,15,0  
    SYNC  
ENDFUNCTION  
  
FUNCTION ModelDetails()  
    PERFORM CHECKLIST FOR OBJECT LIMBS 1  
    SET CURSOR 0,100  
    PRINT "This object contains ", CHECKLIST QUANTITY(),  
    ⤵ " components"  
    FOR c = 1 TO CHECKLIST QUANTITY()  
        PRINT CHECKLIST STRING$(c)  
    NEXT c  
    SYNC  
ENDFUNCTION
```

Activity 35.29

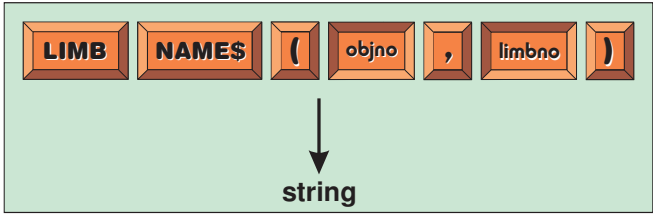
Type in and test the program given in LISTING-35.8 (*limbs08.dbpro*).

The LIMB NAME\$ Statement

When a model is constructed using a 3D package, any limb within that model can be assigned a name. The name of that limb can be found using the LIMB NAME\$ statement. This has the format shown in FIG-35.43.

FIG-35.43

The LIMB NAME\$
Statement



In the diagram:

- objno* is an integer value specifying the object to which the limb in question is attached.
- limbno* is an integer value identifying the limb whose name is to be found.

For example, we could discover the name of limb 2 in model 1 using the line:

```
PRINT LIMB NAME$(1,2)
```

Activity 35.30

Modify the previous program so that the function *ModelDetails()* contains the lines:

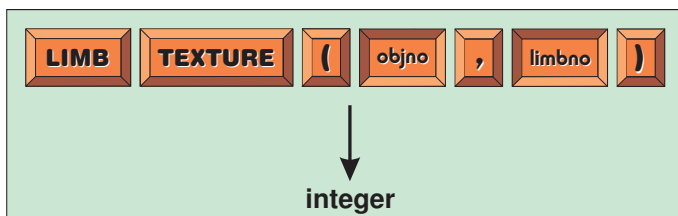
```
FOR c = 1 TO 2
  PRINT LIMB NAME$(1,c)
NEXT c
SYNC
```

The LIMB TEXTURE Statement

If a limb has been textured, the ID of the image used to texture the limb can be discovered using the LIMB TEXTURE statement which has the format shown in FIG-35.44.

FIG-35.44

The LIMB TEXTURE Statement



In the diagram:

objno

is an integer value specifying the object to which the limb in question is attached.

limbno

is an integer value identifying the limb whose texture ID is to be found.

For example, the texture ID for limb 2 of model 1 can be found using a line such as:

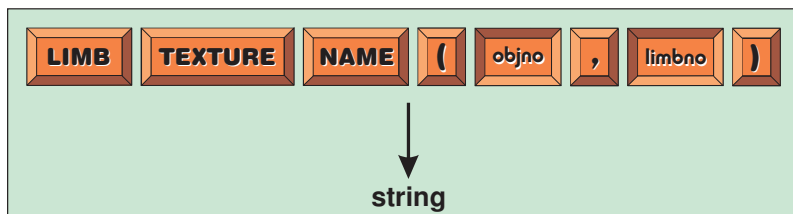
```
limbtextureID = LIMB TEXTURE(1,2)
```

The LIMB TEXTURE NAME Statement

Rather than just get the ID number of a texture, we can retrieve a texture's name using the LIMB TEXTURE NAME statement. Note that the name is not the name of the image file used to texture the limb, but the name assigned to the texture during the building of the model. This means that you'll only get meaningful names when using this statement with models created using a 3D modelling package. The LIMB TEXTURE NAME statement has the format shown in FIG-35.45.

FIG-35.45

The LIMB TEXTURE NAME Statement



In the diagram:

objno

is an integer value specifying the object to which the limb in question is attached.

limbno

is an integer value identifying the limb whose texture name is to be found.

If no name is assigned to the texture, or no texture has been applied to the limb, then an empty string is returned.

For example, the texture name for limb 2 of model 1 can be found using a line such as:

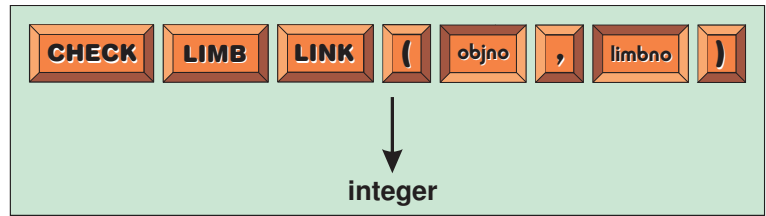
```
texturename$ = LIMB TEXTURE NAME (1,2)
```

The CHECK LIMB LINK Statement

We can check that a new limb can be linked to a specified limb number using the CHECK LIMB LINK statement which has the format shown in FIG-35.46.

FIG-35.46

The CHECK LIMB LINK Statement



In the diagram:

objno

is an integer value specifying the object to which the limb in question is attached.

limbno

is an integer value identifying the limb to which the new limb is to be attached.

The statement returns a value of 1 if a link to the limb specified by *limbno* is legal; otherwise zero is returned.

For example, we could check that a link to limb number 2 in model 1 was possible with code such as:

```
IF CHECK LIMB LINK(1,2) = 1
    PRINT "Link to limb 2 is valid"
ENDIF
```

Saving a Model in DBO Format

Introduction

Once a model has been created, it is possible to save it to a file. The model in that file can then be loaded by another DarkBASIC Pro program. For example, we could create a ladder from a set of cylinders and then save that ladder to a file. The ladder would then be available for use in any other DarkBASIC Pro program.

The DBO File Format

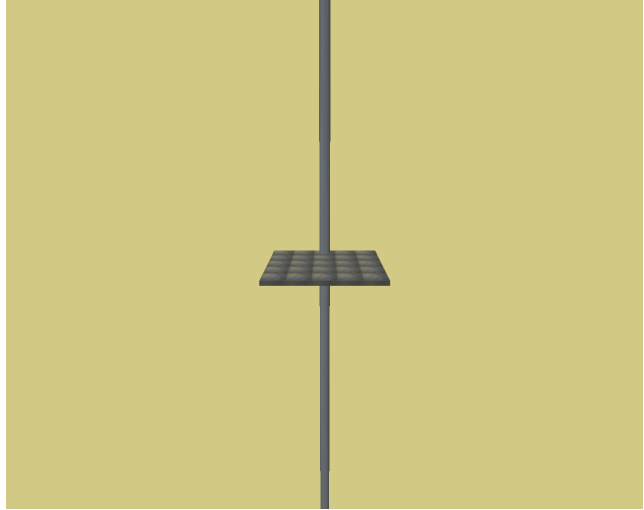
When we save a DarkBASIC Pro model it is stored in DBO (DarkBasic Object) format. This is a text-based file which can be viewed in a normal text editor such as Notepad. A discussion of the structure of this file format is beyond the scope of this text.

Creating an Elevator Model

FIG-35.47 shows the simple platform elevator model we are going to create.

FIG-35.47

The Elevator Model



The model is made from a zero-sized sphere as the root object, with a cylinder and a box as limbs.

The necessary code is given in LISTING-35.9.

LISTING-35.9

Creating a Model for
Saving to a File

```
ScreenSetUp()  
MakeElevator()  
RaiseElevator()  
WAIT KEY  
END  
  
FUNCTION ScreenSetUp()  
    SET DISPLAY MODE 1280,1024,32  
    COLOR BACKDROP RGB(200,200,80)  
    BACKDROP ON  
    AUTOCAM OFF  
    POSITION CAMERA 0,5,-120  
    POINT CAMERA 0,0,0  
ENDFUNCTION  
  
FUNCTION MakeElevator()  
    REM *** Create Objects required ***  
    MAKE OBJECT SPHERE 1,0  
    MAKE OBJECT CYLINDER 2,100  
    SCALE OBJECT 2,0.5,100,0.5  
    MAKE OBJECT BOX 3,5,0.2,5  
    REM *** Create limb meshes ***  
    MAKE MESH FROM OBJECT 1,2  
    MAKE MESH FROM OBJECT 2,3
```

continued on next page

LISTING-35.9
(continued)

Creating a Model for
Saving to a File

The *RaiseElevator()*
function is included only
to show the elevator in
action and is not required
in order to save the
model to a file.

```
REM *** Delete objects ***
DELETE OBJECT 2
DELETE OBJECT 3
REM *** Add limbs to root ***
ADD LIMB 1,1,1
ADD LIMB 1,2,2
REM *** Texture limbs ***
LOAD IMAGE "steel.bmp",1
LOAD IMAGE "metal.bmp",2
TEXTURE LIMB 1,1,1
TEXTURE LIMB 1,2,2
SCALE LIMB TEXTURE 1,2,5,5
REM *** Move platfrom to base ***
OFFSET LIMB 1,2,0,-50,-2
ENDFUNCTION

FUNCTION RaiseElevator()
FOR c = -50 TO 50
    OFFSET LIMB 1,2,0,c,-2
    WAIT 60
NEXT c
ENDFUNCTION
```

Activity 35.31

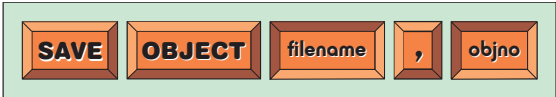
Type in and test the program in LISTING-35.9 (*limbs09.dbpro*).

The SAVE OBJECT Statement

To save an object created in a program to a DBO file, we use the SAVE OBJECT statement which has the format shown in FIG-35.48.

FIG-35.48

The SAVE OBJECT
Statement



In the diagram:

filename

is a string giving the name of the file to be used when saving the model. The filename should be given the extension ".DBO" to correctly identify the type of file being created. The string may include path details.

objno

is an integer value giving the ID of the object to be saved. This can be a single primitive object or a complex model containing limbs. For a limbed model, the root object's ID is used.

For example, to save the elevator model created in the previous program, we would use the line:

```
SAVE OBJECT "elevator.dbo",1
```

Activity 35.32

Replace the call to *RaiseElevator()* in the main section of your last program with the line given above and check that the file *elevator.dbo* is created.

The LOAD OBJECT Statement

The LOAD OBJECT statement is used to load an existing 3D object from a file. This statement can be used to load DBO files, but models stored in other 3D file formats can also be loaded (see Chapter 36).

The LOAD OBJECT statement has the format shown in FIG-35.49.

FIG-35.49

The LOAD OBJECT Statement



In the diagram:

filename

is a string giving the name of the file containing the 3D object. The string may contain drive and folder details.

objno

is the integer value to be assigned to the object loaded from the file. No other object may be assigned the same value.

For example, we could load the elevator model using the line:

```
LOAD OBJECT "elevator.dbo", 1
```

The above line assumes that the file *elevator.dbo* is in the current project's folder.

In fact, this is the simplest form of the LOAD OBJECT statement. We'll leave a full explanation of the statement until the next chapter.

The program in LISTING-35.10 loads the elevator model.

LISTING-35.10

Loading a DBO Model

```
ScreenSetUp()  
LOAD OBJECT "elevator.dbo", 1  
WAIT KEY  
END  
  
FUNCTION ScreenSetUp()  
    SET DISPLAY MODE 1280,1024,32  
    COLOR BACKDROP RGB(200,200,80)  
    BACKDROP ON  
    AUTOCAM OFF  
    POSITION CAMERA 0,5,-20  
    POINT CAMERA 0,0,0  
ENDFUNCTION
```

Activity 35.33

Type in and save the program in LISTING-35.10 (*limbs10.dbpro*).

Copy the *elevator.dbo* file to this project's folder.

Run the program.

What is wrong with the elevator model?

If a model uses textures, the images employed must also be copied to the same folder

as the .DBO file.

Activity 35.34

Copy the *steel.bmp* and *metal.bmp* files used by the elevator model into the *limbs10* folder.

Run the program again.

Is the model correctly textured this time?

Add the *RaiseElevator()* routine to your program and check that the platform moves in the same way as before.

Summary

- A 3D object which is constructed from several component parts is known as a group object or model.
- A model contains a root object and a set of limbs.
- The root object is a standard 3D object.
- The root object may have zero width, height and depth.
- Colouring or texturing the root object will result in the limbs of the model taking on the same colour or texture.
- Meshes are linked to the root object and become limbs within the model.
- Use ADD LIMB to attach a mesh to the root object of a model.
- Limbs are initially positioned with their centre at the centre of the root object.
- Use MAKE OBJECT FROM LIMB to create a normal 3D object from a limb within a model. The limb is unaffected by this operation.
- Use OFFSET LIMB to reposition a limb relative to its root.
- Use ROTATE LIMB to rotate a limb relative to its root.
- Use SCALE LIMB to resize a limb.
- Use COLOR LIMB to colour a specific limb.
- Use TEXTURE LIMB to texture a specific limb.
- Use SCALE LIMB TEXTURE to tile the texture on a specific limb.
- Use SCROLL LIMB TEXTURE to reposition the texture on a specific limb.
- Use HIDE LIMB to make a specific limb invisible.
- Use SHOW LIMB to make a previously invisible limb reappear.
- Use REMOVE LIMB to remove a specific limb from a model.

- Use **LINK LIMB** to link a limb (child limb) to another limb (parent limb) rather than to the root. The child limb's centre is initially positioned at the parent limb's centre.
- Use **CHANGE MESH** to change which mesh is used for an existing limb.
- Use **GLUE OBJECT TO LIMB** to attach an object to a limb.
- Use **UNGLUE OBJECT** to detach a previously glued object from a limb.
- Use **SET LIMB SMOOTHING** to smooth the surface of a specific limb.
- Use **LIMB EXIST** to check if a limb of a specified ID exists.
- Use **LIMB VISIBLE** to check if a specific limb is currently visible.
- Use **LIMB OFFSET** to discover the current offset settings for a specific limb.
- Use **LIMB SCALE** to discover the current scaling factors for a specific limb.
- Use **LIMB ANGLE** to discover a limb's rotation relative to its parent limb or the root.
- Use **LIMB POSITION** to discover a limb's position relative to the world axes.
- Use **LIMB DIRECTION** to discover a limb's rotation relative to the world axes.
- Use **PERFORM CHECKLIST FOR OBJECT LIMBS** to create a list of limbs in a given model.
- Individual limbs can be assigned names if the model was created by a 3D modelling package.
- The root object is included in the list created by the **PERFORM CHECKLIST FOR OBJECT LIMBS**.
- Use **CHECKLIST QUANTITY** and **CHECKLIST STRING\$** to access the contents of the model's checklist.
- Use **LIST NAME\$** to access the name of a specific limb.
- Use **LIMB TEXTURE** to access the image ID of the texture used on a specific limb.
- Use **LIMB TEXTURE NAME** to access the name assigned to the texture used on a specific limb.
- Textures may be named if the 3D model was created by a modelling package.
- Use **CHECK LIMB LINK** to check if a specific limb can be the parent of a new limb.
- Use **SAVE OBJECT** to save primitive shapes or complex models to a file.
- Models saved to a file are stored in DBO format.
- Use **LOAD OBJECT** to load an existing model from a DBO file.

Solutions

Activity 35.1

No solution required.

Activity 35.2

```
REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP 0
BACKDROP ON
REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 0,10,-100
POINT CAMERA 0,0,0
REM *** Make sphere ***
MAKE OBJECT SPHERE 1,10
WAIT KEY
REM *** Make mesh from sphere ***
MAKE MESH FROM OBJECT 1,1
REM *** Delete the original object ***
DELETE OBJECT 1
REM *** Save mesh to file ***
SAVE MESH "sphere01.x",1
REM *** End program ***
WAIT KEY
END
```

```
REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP 0
BACKDROP ON
REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 0,10,-100
POINT CAMERA 0,0,0
REM *** Make sphere ***
MAKE OBJECT SPHERE 1,10
REM *** Texture sphere ***
LOAD IMAGE "marble.bmp",1
TEXTURE OBJECT 1,1
WAIT KEY
REM *** Make mesh from sphere ***
MAKE MESH FROM OBJECT 1,1
REM *** Delete the original object ***
DELETE OBJECT 1
REM *** Save mesh to file ***
SAVE MESH "sphere02.x",1
REM *** End program ***
WAIT KEY
END
```

Activity 35.3

No texture is displayed on the sphere with either .x file.

Activity 35.4

No solution required.

Activity 35.5

```
REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP 0
BACKDROP ON
REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 0,10,-100
POINT CAMERA 0,0,0
REM *** Make and position sphere ***
MAKE OBJECT SPHERE 1,10
```

```
MAKE OBJECT BOX 2,10,10,0.1
REM *** Make mesh from object ***
MAKE MESH FROM OBJECT 1,2
REM *** Delete original object ***
DELETE OBJECT 2
REM *** Attach mesh to object ***
ADD LIMB 1,1,1
REM *** Move object ***
WAIT KEY
POSITION OBJECT 1,-10,0,0
REM *** Allow user to move camera ***
DO
    CONTROL CAMERA USING ARROWKEYS 0,0.1,0.1
LOOP
REM *** End program ***
END
```

When the sphere is moved, any limbs attached to it also move.

Activity 35.6

```
REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP 0
BACKDROP ON
REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 0,10,-100
POINT CAMERA 0,0,0
REM *** Make and position sphere ***
MAKE OBJECT SPHERE 1,10
MAKE OBJECT BOX 2,10,10,0.1
REM *** Make mesh from object ***
MAKE MESH FROM OBJECT 1,2
REM *** Delete original object ***
DELETE OBJECT 2
REM *** Attach mesh to object ***
ADD LIMB 1,1,1
REM *** Move object ***
WAIT KEY
POSITION OBJECT 1,-10,0,0
REM *** Colour root ***
WAIT KEY
COLOR OBJECT 1, RGB(255,0,0)
REM *** Texture root ***
WAIT KEY
LOAD IMAGE "marble.bmp",1
TEXTURE OBJECT 1,1
REM *** Allow user to move camera ***
DO
    CONTROL CAMERA USING ARROWKEYS 0,0.1,0.1
LOOP
REM *** End program ***
END
```

Any colour or texture applied to the root object is also applied to the limbs.

Activity 35.7

```
REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP 0
BACKDROP ON
REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 0,10,-100
POINT CAMERA 0,0,0
REM *** Make and position sphere ***
MAKE OBJECT SPHERE 1,10
MAKE OBJECT BOX 2,10,10,0.1
REM *** Make mesh from object ***
```

```

MAKE MESH FROM OBJECT 1,2
REM *** Delete original object ***
DELETE OBJECT 2
REM *** Attach mesh to object ***
ADD LIMB 1,1,1
REM *** Move object ***
WAIT KEY
POSITION OBJECT 1,-10,0,0
REM *** Colour root ***
WAIT KEY
COLOR OBJECT 1, RGB(255,0,0)
REM *** Texture root ***
WAIT KEY
LOAD IMAGE "marble.bmp",1
TEXTURE OBJECT 1,1
REM *** Create an object from the limb ***
WAIT KEY
MAKE OBJECT FROM LIMB 2,1,1
REM *** Reposition new object ***
POSITION OBJECT 2,20,0,0
REM *** Allow user to move camera ***
DO
    CONTROL CAMERA USING ARROWKEYS 0,0.1,0.1
LOOP
REM *** End program ***
END

```

Activity 35.8

The limb moves to the far edge of the sphere.

Activity 35.9

```

REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP 0
BACKDROP ON
REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 0,10,-100
POINT CAMERA 0,0,0
REM *** Make and position sphere ***
MAKE OBJECT SPHERE 1,10
MAKE OBJECT BOX 2,10,10,0.1
REM *** Make mesh from object ***
MAKE MESH FROM OBJECT 1,2
REM *** Delete original object ***
DELETE OBJECT 2
REM *** Attach mesh to object ***
ADD LIMB 1,1,1
REM *** Move object ***
WAIT KEY
POSITION OBJECT 1,-10,0,0
REM *** Colour root ***
WAIT KEY
COLOR OBJECT 1, RGB(255,0,0)
REM *** Texture root ***
WAIT KEY
REM *** Add second limb ***
ADD LIMB 1,2,1
OFFSET LIMB 1,2,0,0,-5
LOAD IMAGE "marble.bmp",1
TEXTURE OBJECT 1,1
REM *** Allow user to move camera ***
DO
    CONTROL CAMERA USING ARROWKEYS 0,0.1,0.1
LOOP
REM *** End program ***
END

```

Activity 35.10

```

REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP 0

```

```

BACKDROP ON
REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 0,10,-100
POINT CAMERA 0,0,0
REM *** Make and position sphere ***
MAKE OBJECT SPHERE 1,10
MAKE OBJECT BOX 2,10,10,0.1
REM *** Make mesh from object ***
MAKE MESH FROM OBJECT 1,2
REM *** Delete original object ***
DELETE OBJECT 2
REM *** Attach mesh to object ***
ADD LIMB 1,1,1
OFFSET LIMB 1,1,0,0,5
ADD LIMB 1,2,1
OFFSET LIMB 1,2,0,0,-5
ADD LIMB 1,3,1
ROTATE LIMB 1,3,90,0,0
OFFSET LIMB 1,3,0,-5,0
ADD LIMB 1,4,1
ROTATE LIMB 1,4,90,0,0
OFFSET LIMB 1,4,0,5,0
ADD LIMB 1,5,1
ROTATE LIMB 1,5,0,90,0
OFFSET LIMB 1,5,-5,0,0
ADD LIMB 1,6,1
ROTATE LIMB 1,6,0,90,0
OFFSET LIMB 1,6,5,0,0
REM *** Rotate model ***
DO
    PITCH OBJECT UP 1,1
    TURN OBJECT LEFT 1,1
    ROLL OBJECT LEFT 1,1
    WAIT 20
LOOP
REM *** End program ***
END

```

Activity 35.11

The first version requires the line

```
SCALE LIMB 1,4, 200,200,100
```

The second version of the program is shown below:

```

REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP 0
BACKDROP ON
REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 0,10,-100
POINT CAMERA 0,0,0
REM *** Make and position sphere ***
MAKE OBJECT SPHERE 1,10
MAKE OBJECT BOX 2,10,10,0.1
REM *** Make mesh from object ***
MAKE MESH FROM OBJECT 1,2
REM *** Delete original object ***
DELETE OBJECT 2
REM *** Attach mesh to object ***
ADD LIMB 1,1,1
OFFSET LIMB 1,1,0,0,5
ADD LIMB 1,2,1
OFFSET LIMB 1,2,0,0,-5
ADD LIMB 1,3,1
ROTATE LIMB 1,3,90,0,0
OFFSET LIMB 1,3,0,-5,0
ADD LIMB 1,4,1
ROTATE LIMB 1,4,90,0,0
OFFSET LIMB 1,4,0,5,0
ADD LIMB 1,5,1
ROTATE LIMB 1,5,0,90,0
OFFSET LIMB 1,5,-5,0,0

```

```

ADD LIMB 1,6,1
ROTATE LIMB 1,6,0,90,0
OFFSET LIMB 1,6,5,0,0
REM *** Scale limb 4 ***
DO
  FOR scale = 100 TO 200
    SCALE LIMB 1,4,scale,scale,100
    WAIT 10
  NEXT scale
  FOR scale = 200 TO 100 STEP -1
    SCALE LIMB 1,4,scale,scale,100
    WAIT 10
  NEXT scale
LOOP
REM *** End program ***
END

```

Activity 35.12

```

REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP 0
BACKDROP ON
REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 0,10,-100
POINT CAMERA 0,0,0
REM *** Make and position sphere ***
MAKE OBJECT SPHERE 1,10
MAKE OBJECT BOX 2,10,10,0.1
REM *** Make mesh from object ***
MAKE MESH FROM OBJECT 1,2
REM *** Delete original object ***
DELETE OBJECT 2
REM *** Attach 6 limbs ***
ADD LIMB 1,1,1
OFFSET LIMB 1,1,0,0,5
ADD LIMB 1,2,1
OFFSET LIMB 1,2,0,0,-5
ADD LIMB 1,3,1
ROTATE LIMB 1,3,90,0,0
OFFSET LIMB 1,3,0,-5,0
ADD LIMB 1,4,1
ROTATE LIMB 1,4,90,0,0
OFFSET LIMB 1,4,0,5,0
ADD LIMB 1,5,1
ROTATE LIMB 1,5,0,90,0
OFFSET LIMB 1,5,-5,0,0
ADD LIMB 1,6,1
ROTATE LIMB 1,6,0,90,0
OFFSET LIMB 1,6,5,0,0
REM *** Colour limbs ***
COLOR LIMB 1,1,RGB(255,255,0)
COLOR LIMB 1,2,RGB(255,0,0)
COLOR LIMB 1,3,RGB(0,0,255)
COLOR LIMB 1,4,RGB(255,0,255)
COLOR LIMB 1,5,RGB(0,255,255)
COLOR LIMB 1,6,RGB(0,255,0)
REM *** rotate model ***
DO
  PITCH OBJECT UP 1,1
  TURN OBJECT LEFT 1,1
  ROLL OBJECT LEFT 1,1
  WAIT 20
LOOP
REM *** End program ***
END

```

Activity 35.13

No solution required.

Activity 35.14

```
REM *** Set up screen ***
```

```

SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP 0
BACKDROP ON
REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 0,10,-100
POINT CAMERA 0,0,0
REM *** Make and position sphere ***
MAKE OBJECT SPHERE 1,10
MAKE OBJECT BOX 2,10,10,0.1
REM *** Make mesh from object ***
MAKE MESH FROM OBJECT 1,2
REM *** Delete original object ***
DELETE OBJECT 2
REM *** Attach 6 limbs ***
ADD LIMB 1,1,1
OFFSET LIMB 1,1,0,0,5
ADD LIMB 1,2,1
OFFSET LIMB 1,2,0,0,-5
ADD LIMB 1,3,1
ROTATE LIMB 1,3,90,0,0
OFFSET LIMB 1,3,0,-5,0
ADD LIMB 1,4,1
ROTATE LIMB 1,4,90,0,0
OFFSET LIMB 1,4,0,5,0
ADD LIMB 1,5,1
ROTATE LIMB 1,5,0,90,0
OFFSET LIMB 1,5,-5,0,0
ADD LIMB 1,6,1
ROTATE LIMB 1,6,0,90,0
OFFSET LIMB 1,6,5,0,0
REM *** Colour all limbs blue ***
FOR limbno = 1 TO 6
  COLOR LIMB 1,limbno,RGB(0,0,255)
NEXT limbno
REM *** Display video on limb 2 ***
LOAD ANIMATION "vd.mpg",1
PLAY ANIMATION TO IMAGE 1,1,0,0,120,80
TEXTURE LIMB 1,2,1
REM *** rotate model ***
DO
  PITCH OBJECT UP 1,1
  TURN OBJECT LEFT 1,1
  ROLL OBJECT LEFT 1,1
  WAIT 20
LOOP
REM *** End program ***
END

```

Activity 35.15

```

REM *** Main program ***
SetUpScreen()
CreateDiceShape()
TextureDice()
RotateDice()
SCALE LIMB TEXTURE 1,1,2,3
REM *** End program ***
WAIT KEY
END

FUNCTION SetUpScreen()
  REM *** Set up screen ***
  SET DISPLAY MODE 1280,1024,32
  COLOR BACKDROP RGB(0,200,0)
  BACKDROP ON
  REM *** Position camera ***
  AUTOCAM OFF
  POSITION CAMERA 0,20,-100
  POINT CAMERA 0,0,0
ENDFUNCTION

FUNCTION CreateDiceShape()
  REM *** Make and position sphere ***
  MAKE OBJECT SPHERE 1,10
  MAKE OBJECT BOX 2,10,10,0.1
  REM *** Make mesh from object ***

```

```

MAKE MESH FROM OBJECT 1,2
REM *** Delete original object ***
DELETE OBJECT 2
REM *** Attach back ***
ADD LIMB 1,1,1
OFFSET LIMB 1,1,0,0,5
REM *** Add front ***
ADD LIMB 1,2,1
OFFSET LIMB 1,2,0,0,-5
REM *** Add bottom ***
ADD LIMB 1,3,1
ROTATE LIMB 1,3,90,0,0
OFFSET LIMB 1,3,0,-5,0
REM *** Add top ***
ADD LIMB 1,4,1
ROTATE LIMB 1,4,90,0,0
OFFSET LIMB 1,4,0,5,0
REM *** Add left side ***
ADD LIMB 1,5,1
ROTATE LIMB 1,5,0,90,0
OFFSET LIMB 1,5,-5,0,0
REM *** Add right side ***
ADD LIMB 1,6,1
ROTATE LIMB 1,6,0,90,0
OFFSET LIMB 1,6,5,0,0
ENDFUNCTION

FUNCTION TextureDice()
REM *** Load Images ***
LOAD IMAGE "Spot1.bmp", 1
LOAD IMAGE "Spot2.bmp", 2
LOAD IMAGE "Spot3.bmp", 3
LOAD IMAGE "Spot4.bmp", 4
LOAD IMAGE "Spot5.bmp", 5
LOAD IMAGE "Spot6.bmp", 6
REM *** Colour top red ***
TEXTURE LIMB 1,1,1
TEXTURE LIMB 1,2,6
TEXTURE LIMB 1,3,2
TEXTURE LIMB 1,4,5
TEXTURE LIMB 1,5,3
TEXTURE LIMB 1,6,4
ENDFUNCTION

FUNCTION RotateDice()
XROTATE OBJECT 1,180
ENDFUNCTION

```

```

#CONSTANT rootobj 1
#CONSTANT planeobj 2
REM *** Meshes ***
#CONSTANT planemesh 1
REM *** Limbs ***
#CONSTANT floorlimb 1
#CONSTANT ceilinglimb 2
#CONSTANT wall1limb 3
#CONSTANT wall2limb 4
REM *** Load texture images ***
LOAD IMAGE "floor_m_01.bmp", floorimg
LOAD IMAGE "ceil_m_03.bmp", ceilingimg
LOAD IMAGE "wall_m_44.bmp", wallimg
REM *** Create plane mesh ***
MAKE OBJECT BOX planeobj,10,200,0.1
MAKE MESH FROM OBJECT planemesh,planeobj
DELETE OBJECT planeobj
REM *** Create corridor model ***
REM *** Make invisible root ***
MAKE OBJECT SPHERE rootobj,0
REM *** Add floor ***
ADD LIMB rootobj,floorlimb,planemesh
TEXTURE LIMB rootobj,floorlimb,ceilingimg
SCALE LIMB TEXTURE rootobj,floorlimb,2,40
ROTATE LIMB rootobj,floorlimb,90,0,0
ADD LIMB rootobj,ceilinglimb,planemesh
TEXTURE LIMB rootobj,ceilinglimb,ceilingimg
SCALE LIMB TEXTURE rootobj,ceilinglimb,2,40
ROTATE LIMB rootobj,ceilinglimb,90,0,0
OFFSET LIMB rootobj,ceilinglimb,0,10,0
ADD LIMB rootobj,wall1limb,planemesh
TEXTURE LIMB rootobj,wall1limb,wallimg
SCALE LIMB TEXTURE rootobj,wall1limb,2,40
ROTATE LIMB rootobj,wall1limb,90,0,90
OFFSET LIMB rootobj,wall1limb,5,5,0
ADD LIMB rootobj,wall2limb,planemesh
TEXTURE LIMB rootobj,wall2limb,wallimg
SCALE LIMB TEXTURE rootobj,wall2limb,2,40
ROTATE LIMB rootobj,wall2limb,90,0,90
OFFSET LIMB rootobj,wall2limb,-5,5,0
HIDE LIGHT 0
ENDFUNCTION

```

```

FUNCTION RotateDice()
XROTATE OBJECT 1,180
ENDFUNCTION

```

Activity 35.17

The main section of the program should be changed to

```

REM *** Main section ***
SetUpScreen()
CreateModel()
REM *** Give user control of camera ***
DO
CONTROL CAMERA USING ARROWKEYS 0,0.1,0.1
LOOP
REM *** End program ***
END

FUNCTION SetUpScreen()
SET DISPLAY MODE 1280, 1024,32

COLOR BACKDROP RGB(25,25,25)
BACKDROP ON
AUTOCAM OFF
POSITION CAMERA 0,4,-100
POINT CAMERA 0,0,0
ENDFUNCTION

FUNCTION CreateModel()
REM *** Declare named constants ***
REM *** Images ***
#CONSTANT floorimg 1
#CONSTANT ceilingimg 2
#CONSTANT wallimg 3
REM *** Objects ***

```

```

REM *** Main program ***
SetUpScreen()
CreateDiceShape()
TextureDice()
RotateDice()
HIDE LIMB 1,1
REM *** End program ***
WAIT KEY
END

```

The resulting model is shown below:



Activity 35.18

In the function *CreateDiceShape()*, the line

```
MAKE OBJECT SPHERE 1,10
```

should be changed to

```
MAKE OBJECT SPHERE 1,0
```

The effect of this change on the model displayed on screen is shown below



Activity 35.19

```

REM *** Main program ***
SetUpScreen()
CreateDiceShape()
TextureDice()
RotateDice()
HIDE LIMB 1,1
REM *** Make limb reappear ***
WAIT KEY
SHOW LIMB 1,1
REM *** End program ***
WAIT KEY
END

```

Activity 35.20

Modify the main section to

```

REM *** Main program ***
SetUpScreen()
CreateModel()
ManipulateModel()
REM *** Remove cylinder ***
WAIT KEY
REMOVE LIMB 1,1
REM *** End program ***
WAIT KEY
END

```

Since the cone is attached to the cylinder, it is also removed from the model.

Activity 35.21

Modify the main section to:

```

REM *** Main program ***
SetUpScreen()
CreateModel()
ManipulateModel()
REM *** Change cone to sphere ***
WAIT KEY
MAKE OBJECT SPHERE 2,5
MAKE MESH FROM OBJECT 3,2
CHANGE MESH 1,2,3
REM *** End program ***
WAIT KEY
END

```

Activity 35.22

```

REM *** Main program ***
SetUpScreen()
CreateModel()
DemoGlue()
REM *** End program ***
WAIT KEY
END

```

```

FUNCTION SetUpScreen()
REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP RGB(120,67,39)
BACKDROP ON
REM *** Set up camera ***
AUTOCAM OFF
POSITION CAMERA 0,0,-100
POINT CAMERA 0,0,0
ENDFUNCTION

```

```

FUNCTION CreateModel()
REM *** Make model root ***
MAKE OBJECT SPHERE 1,8
REM *** Create objects used as limbs ***
MAKE OBJECT CYLINDER 2,10
SCALE OBJECT 2, 20,300,20
MAKE OBJECT CONE 3,5
REM *** Convert objects to meshes ***
MAKE MESH FROM OBJECT 1,2
MAKE MESH FROM OBJECT 2,3
REM *** Delete limb objects ***
DELETE OBJECT 2
DELETE OBJECT 3
REM *** Add limbs to root ***
ADD LIMB 1,1,1
ADD LIMB 1,2,2
REM *** Make cone child of cylinder ***
LINK LIMB 1,1,2
REM *** Move cone to end of cylinder ***
OFFSET LIMB 1,2,0,15,0
REM *** Position cylinder ***
OFFSET LIMB 1,1,0,15,0
ENDFUNCTION

```

```

FUNCTION DemoGlue()
REM *** Create and position cube ***
MAKE OBJECT CUBE 4,3
POSITION OBJECT 4,30,0,0
REM *** Rotate model to cube ***
FOR angle = 0 TO 85
ROLL OBJECT RIGHT 1,1
WAIT 50
NEXT angle
REM *** Glue cube to cone ***
GLUE OBJECT TO LIMB 4,1,2,0
WAIT KEY
REM *** Rotate model upright ***
FOR angle = 0 TO 85
ROLL OBJECT LEFT 1,1
WAIT 50
NEXT angle
ENDFUNCTION

```

When the program runs, the cube takes on the same offset and rotational values as the cone.

A *mode* value of 1 gives the same effect, but with *mode* set to 2, the centre of the cube is linked to the centre of the cone.

Activity 35.23

The code for *DemoGlue()* changes to:

```
FUNCTION DemoGlue()  
  REM *** Create and position cube ***  
  MAKE OBJECT CUBE 4,3  
  POSITION OBJECT 4,30,0,0  
  REM *** Rotate model to cube ***  
  FOR angle = 0 TO 85  
    ROLL OBJECT RIGHT 1,1  
    WAIT 50  
  NEXT angle  
  REM *** Glue cube to cone ***  
  GLUE OBJECT TO LIMB 4,1,2,2  
  POSITION OBJECT 4,2,0,0  
  WAIT KEY  
  REM *** Rotate model upright ***  
  FOR angle = 0 TO 85  
    ROLL OBJECT LEFT 1,1  
    WAIT 50  
  NEXT angle  
ENDFUNCTION
```

Activity 35.24

DemoGlue() should be coded as:

```
FUNCTION DemoGlue()  
  REM *** Create and position cube ***  
  MAKE OBJECT CUBE 4,3  
  POSITION OBJECT 4,30,0,0  
  REM *** Rotate model to cube ***  
  FOR angle = 0 TO 85  
    ROLL OBJECT RIGHT 1,1  
    WAIT 50  
  NEXT angle  
  REM *** Glue cube to cone ***  
  GLUE OBJECT TO LIMB 4,1,2,2  
  POSITION OBJECT 4,2,0,0  
  WAIT KEY  
  REM *** Rotate model upright ***  
  FOR angle = 0 TO 85  
    ROLL OBJECT LEFT 1,1  
    WAIT 50  
  NEXT angle  
  REM *** Model back to "ground" level ***  
  WAIT KEY  
  FOR angle = 0 TO 85  
    ROLL OBJECT RIGHT 1,1  
    WAIT 50  
  NEXT angle  
  REM *** Release cube ***  
  WAIT KEY  
  UNGLUE OBJECT 4  
  REM *** Model back to start position ***  
  WAIT KEY  
  FOR angle = 0 TO 85  
    ROLL OBJECT LEFT 1,1  
    WAIT 50  
  NEXT angle  
ENDFUNCTION
```

When the cube is released it repositions itself at the centre of the cone. We could use POSITION OBJECT to return it to its original position.

Activity 35.25

The main section should be coded as:

```
REM *** Main program ***  
SetUpScreen()  
CreateModel()
```

```
REM *** Smooth cone ***  
WAIT KEY  
SET LIMB SMOOTHING 1,2,100  
`DemoGlue()  
REM *** End program ***  
WAIT KEY  
END
```

Activity 35.26

The new version of *SwingDoor()*:

```
FUNCTION SwingDoor(angle, delay)  
  REM *** Get the current door angle ***  
  currentangle =  
  OBJECT ANGLE Y(doorhingeobj)  
  REM *** Check direction of swing ***  
  IF angle > currentangle  
    incr = 1  
  ELSE  
    incr = -1  
  ENDIF  
  REM *** Swing from starting angle ***  
  REM *** in 1 degree steps ***  
  FOR c = currentangle TO angle STEP incr  
    ROTATE OBJECT doorhingeobj,0,c,0  
    WAIT delay  
  NEXT c  
ENDFUNCTION
```

We could also modify the routine to add checks on the parameters to make sure *angle* and *delay* lie within acceptable limits.

To test the function, we could change the main section to:

```
SetupScreen()  
CreateDoor()  
WAIT KEY  
SwingDoor(20,50)  
WAIT KEY  
SwingDoor(80,50)  
WAIT KEY  
SwingDoor(45,50)  
WAIT KEY  
END
```

If we swing the door too far, we'll see that the hinges and handle are on the wrong sides on the reverse of the door! We could cure this by making the door itself a group object.

Activity 35.27

To see the offsets for each limb the main section could be coded as:

```
REM *** Main program ***  
SetUpScreen()  
CreateModel()  
DO  
  SET CURSOR 100,100  
  PRINT "Cylinder offset : ",  
    ↵LIMB OFFSET X(1,1),":",  
    ↵LIMB OFFSET Y(1,1),":",  
    ↵LIMB OFFSET Z(1,1)  
  SET CURSOR 100,130  
  PRINT "Cone offset : ",  
    ↵LIMB OFFSET X(1,2),":",  
    ↵LIMB OFFSET Y(1,2),":",  
    ↵LIMB OFFSET Z(1,2)  
LOOP  
REM *** End program ***  
END
```


Activity 35.28

No solution required.

```
FUNCTION RaiseElevator()  
  FOR c = -50 TO 50  
    OFFSET LIMB 1,2,0,c,-2  
    WAIT 60  
  NEXT c  
ENDFUNCTION
```

Activity 35.29

No solution required.

Activity 35.30

The updated version of *Modeldetails()* should be:

```
FUNCTION ModelDetails()  
  PERFORM CHECKLIST FOR OBJECT LIMBS 1  
  PRINT "This object contains ",  
  CHECKLIST QUANTITY(), " components"  
  FOR c = 1 TO CHECKLIST QUANTITY()  
    PRINT CHECKLIST STRING$(c)  
  NEXT c  
  SYNC  
  FOR c = 1 To 2  
    PRINT LIMB NAME$(1,c)  
  NEXT c  
  SYNC  
ENDFUNCTION
```

The limbs are titled *new limb* in each case. We would have to create the model in an external package to allocate names to the limbs.

Activity 35.31

No solution required.

Activity 35.32

The main section of your program should now be as follows:

```
ScreenSetUp()  
MakeElevator()  
SAVE OBJECT "elevator.dbo",1  
WAIT KEY  
END
```

Activity 35.33

The model is not textured.

Activity 35.34

The model is correctly textured when the required images are placed in the project folder.

The final version of the program should be:

```
ScreenSetUp()  
LOAD OBJECT "elevator.dbo",1  
RaiseElevator()  
WAIT KEY  
END  
FUNCTION ScreenSetUp()  
  SET DISPLAY MODE 1280,1024,32  
  COLOR BACKDROP RGB(200,200,80)  
  BACKDROP ON  
  AUTOCAM OFF  
  POSITION CAMERA 0,5,-120  
  POINT CAMERA 0,0,0  
ENDFUNCTION
```


Importing 3D Objects

3D File Formats

How to Load Existing 3D Objects

Frame Interpolation

Joining Animated Models

Playing Animated Models

Setting the Animation Speed

Importing 3D Objects

Introduction

Creating 3D objects using DarkBASIC Pro is all very well, but ultimately is probably too limiting to allow us to achieve the sophistication we need in most games. For example, it would be rather difficult to create a 3D gun, never mind a humanoid, using only spheres, boxes, and cylinders.

Realistically, we need to create these complex shapes in some other package - and there are many packages out there which are designed specifically to do just that. If you've already made a few million from selling your DarkBASIC Pro game, then you can afford professional-level 3D creation software such as Maya, 3D Studio Max, or XSI SoftImage. For those with a more limited budget, have a look at some of the packages available from The Game Creators such as 3D Canvas Pro or gameSpace. There are even free or shareware packages such as Anim8or and Milkshape.

Don't worry if you have absolutely no artistic ability (like me!), you'll still be able to create models with 3D modelling software that will truly amaze you; if you do have an artistic side, then you will be able to produce just about anything you can imagine.

One word of warning, when creating 3D models that are to be used in a game, you need to keep the number of polygons used as low as possible. The more polygons an object contains, the more load you place on the processor and graphics card. With more processing to do, the game's frame rate will drop, and when it gets too low, your game will become jerky and unconvincing.

As usual, there is software out there to help overcome even this problem. Polygon-reducing software attempts to reduce the number of polygons a 3D object uses, while still retaining as much detail as possible.

Of course, learning how to use a 3D modelling package takes time - a long time if you want to be really proficient - so, as an alternative, you can buy ready-to-use 3D models such as the Dark Matter series offered by The Game Creators (see FIG-36.1).

FIG-36.1

A 3D Model from the
Dark Matter CD



It is not within the scope of this book to show you how to use a 3D modelling package, but if you would like to have a go at creating your own objects - and you should - you'll find plenty of tutorial material on the Internet for most well-known packages.

File Formats

Almost every 3D modelling package has its own format for saving objects to a file. DarkBASIC Pro accepts several formats, BSP, MD2, MD3, MDL, PAK , but the commonest are Microsoft's DirectX format (these files end with a *.x* extension) and 3D Studio format (*.3ds* extension) as well as DarkBASIC Pro's own *.dbo* format.

The examples given in this chapter make use of files available from Dark Matter Volume 1.

Statements for Loading and Using 3D Objects

The LOAD OBJECT Statement Again

In the last chapter we saw that the LOAD OBJECT statement can be used to load an existing 3D model stored in DBO format. However, the same instruction can be used to load *.x* and *.3ds* files created by separate modelling packages. The extended syntax for the LOAD OBJECT statement is shown in FIG-36.2.

FIG-36.2 The LOAD OBJECT Statement



In the diagram:

filename

is a string giving the name of the file containing the model to be loaded.

objno

is an integer value giving the ID to be assigned to the model once it is loaded.

loadmode

is an integer value specifying how texture and other surface effects are to be handled.

Possible values are:

- 0 - DarkBASIC default method
- 1 - DarkBASIC Pro default method
- 2 - Retain material/diffuse effects
- 3 - Retain material/texture effects
- 4 - Blend texture/diffuse effects
- 5 - Retain multi-material effects

reducemode

is an integer value used to scale loaded texture images.

x represents any positive integer value.

- 0 - No scaling
- x* - Original size divided by *x*

The program in LISTING-36.1 demonstrates how to load the high resolution version of an AK47 rifle, available in Dark Matter 1. The object is then rotated.

LISTING-36.1

Loading a Static Model

```
REM *** Set screen resolution ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP RGB(155,155,0)
BACKDROP ON

REM *** Load the gun object ***
LOAD OBJECT "H-AK47-Static.X",1
REM *** Position camera ***
POSITION CAMERA 0,0,-2
REM *** rotate gun ***
DO
    TURN OBJECT LEFT 1,1
    PITCH OBJECT UP 1,1
    WAIT 10
LOOP
REM *** End program ***
END
```

Activity 36.1

Type in and test the program in LISTING-36.1 (*loaded01.dbpro*). Make sure you've copied the .x file used into the program's directory.

Although the gun is the correct shape, it doesn't have any texturing. The weapon has been designed with a texture, but this is held in a separate file called *AK47.DDS*. If we want the texture to appear on the gun, we must copy the .dds file to the current program folder.

Activity 36.2

By clicking on DarkBASIC Pro's **Media** button and selecting **Add**, copy the *AK47.DDS* file into the *loaded01* folder created by the last program.

Re-run your program. (The gun should now be displaying its texture.)

Activity 36.3

Modify your last program to load and rotate the jet fighter held in file *H-Jet Fighter-Move.x*. The associated textures are held in *JetFighter.dds*.

Once loaded, a 3D object can be manipulated using all the standard OBJECT statements, thus allowing us to do such things as position, rotate and resize objects.

Not all 3D objects are static, unmoving things. Animated 3D objects will actually be held as a series of frames; a bit like an animated sprite. For example, Dark Matter's Hive-Brain alien comes in several files, each of these contain a few frames showing the alien performing some basic movement.

The program in LISTING-36.2 loads the walking alien.

LISTING-36.2

Loading an Animated Image

```
REM *** Set screen resolution and backdrop ***
SET DISPLAY MODE 1280, 1024, 32
COLOR BACKDROP 0
BACKDROP ON
REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 0,1,-10
```

continued on next page

LISTING-36.2
(continued)

Loading an Animated
Image

```
REM *** Create a point light ***
SET POINT LIGHT 0,-10,10,0
SET LIGHT RANGE 0,15

REM *** Load object and position camera ***
LOAD OBJECT "H-Alien Hivebrain-Move.X",1

REM *** End program ***
WAIT KEY
END
```

Activity 36.4

Type in and test the program given in LISTING-36.2 (*loaded02.dbpro*).

Despite being an animated model, the alien shows no sign of any movement.

The PLAY OBJECT Statement

To get the object to move, we need to play each frame in the animation and this is done using the PLAY OBJECT statement. The statement offers to play every frame in the animation or just a specified set of frames. The PLAY OBJECT statement has the format shown in FIG-36.3.

FIG-36.3

The PLAY OBJECT
Statement



In the diagram:

- objno* is an integer value specifying the 3D object to be played. Only objects which consist of two or more frames can be specified.
- start* is an integer value specifying the first frame to be played. The first frame of an animation is frame zero.
- finish* is an integer value specifying the last frame to be played.

If *start* and *finish* are omitted, the whole animation is played. If *finish* is omitted every frame from *start* to the end of the animation is played.

For example,

```
PLAY OBJECT 1
```

will play every frame in object 1 while

```
PLAY OBJECT 1,0,10
```

will play frames 0 to 10 only and

```
PLAY OBJECT 1,5
```

will play from frame 5 to the end of the animation.

Activity 36.5

In your last program, add the lines

WAIT KEY

PLAY OBJECT 1

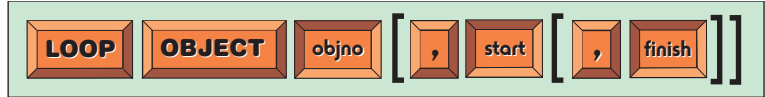
immediately after the **LOAD OBJECT** statement.

The LOOP OBJECT Statement

If we want the frames of an animated 3D object to play repeatedly, then we need to use the **LOOP OBJECT** statement which has the format shown in FIG-36.4.

FIG-36.4

The **LOOP OBJECT**
Statement



In the diagram:

objno

is an integer value specifying the 3D object to be looped. Only objects which consist of two or more frames can be specified.

start

is an integer value specifying the first frame to be included in the loop.

finish

is an integer value specifying the last frame to be included in the loop.

If *start* and *finish* are omitted, the whole animation is played. If *finish* is omitted every frame from *start* to the end of the animation is played.

Activity 36.6

In your program, change the **PLAY OBJECT** statement to **LOOP OBJECT** and re-run the program.

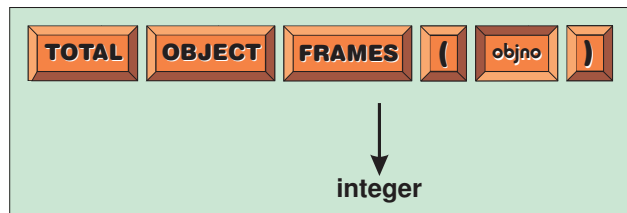
To create a smooth animation, the first and last frames need to be almost identical. The walking alien judders slightly as it cycles back to the first frame. We'll improve this in a moment.

The TOTAL OBJECT FRAMES Statement

If we didn't create the animated 3D object ourselves, then we need to find out just how many frames are in the animation. We can do this using the **TOTAL OBJECT FRAMES** statement which returns the number of frames in a specified object. The statement has the format shown in FIG-36.5.

FIG-36.5

The **TOTAL OBJECT**
FRAMES Statement



In the diagram:

objno

is an integer value specifying the 3D object whose frame count is to be returned.

Activity 36.7

Add the appropriate code to your program so that the number of frames in the alien animation is displayed on the screen.

Now that we know just how many frames are in the animated object, we can limit exactly which frames are played.

Activity 36.8

Modify the LOOP OBJECT statement in your program so that only frames 0 to 20 are used in the animation shown on the screen.

Moving the Alien

Our alien is walking on the spot, but it would be more impressive if it actually went somewhere. In LISTING-36.3 the alien walks towards camera and eventually disappears as it passes the camera.

LISTING-36.3

Moving an Animated Object

```
REM *** Set screen resolution ***
SET DISPLAY MODE 1280, 1024, 32
COLOR BACKDROP 0
BACKDROP ON
AUTOCAM OFF
REM *** Create a point light ***
SET POINT LIGHT 0,-10,10,0
SET LIGHT RANGE 0,15
REM *** Load object; position camera ***
LOAD OBJECT "H-Alien Hivebrain-Move.X",1
POSITION CAMERA 0,1,-10
REM *** Play object continuously ***
LOOP OBJECT 1,0,20
REM *** Move alien ***
DO
    MOVE OBJECT 1,-0.03
LOOP
REM *** End program ***
END
```

Activity 36.9

Type in and test the program in LISTING-36.3 (*loaded03.dbpro*).

The SET OBJECT SPEED Statement

The speed at which the frames of an animated 3D object are played can be set using the SET OBJECT SPEED statement which has the format shown in FIG-36.6.

FIG-36.6

The SET OBJECT SPEED Statement



In the diagram:

<i>objno</i>	is an integer value giving the ID of the 3D object whose speed is to be set.
<i>perc</i>	is an integer value giving the new speed as a percentage of normal speed. For example, a value of 50 will play the frames at half normal speed, while 200 gives twice normal speed.

Activity 36.10

In *loaded03.dbpro*, change the speed of the alien's walk to 50% of normal.

The STOP OBJECT Statement

A playing or looping animated object can be stopped using the STOP OBJECT statement which has the format shown in FIG-36.7.

FIG-36.7

The STOP OBJECT Statement



In the diagram:

<i>objno</i>	is an integer value giving the ID of the 3D object whose animation is to be stopped.
--------------	--

When stopped, the model will continue to show the last frame played. This could be any frame within the animation.

Activity 36.11

Modify your program so that the hive alien's animation (and repositioning) is halted when it moves to a z-ordinate value of less than -6.

The SET OBJECT FRAME Statement

When an animated model is first loaded, it displays frame zero. We can set the model to show any specific frame using the SET OBJECT FRAME statement which has the format shown in FIG-36.8.

FIG-36.8

The SET OBJECT FRAME Statement



In the diagram:

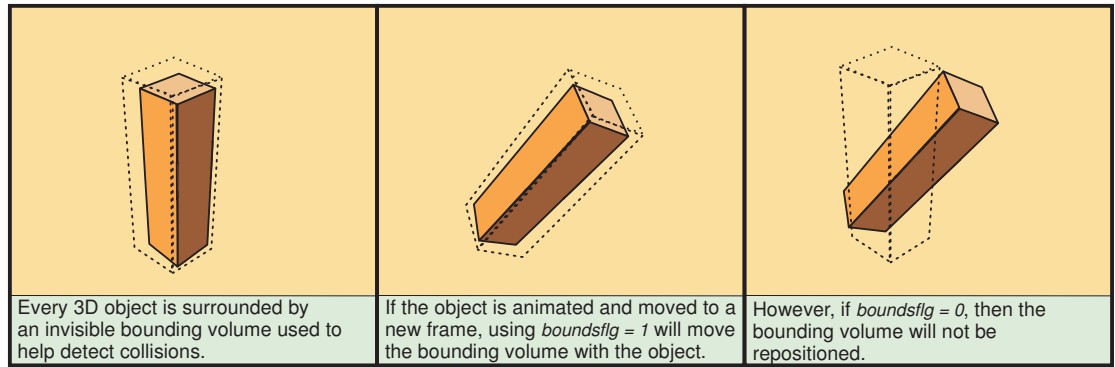
<i>objno</i>	is an integer value giving the ID of the 3D object to which this operation is to be applied.
<i>frame</i>	is an integer value specifying the frame of the 3D animated object to be displayed.
<i>boundsflg</i>	0 or 1.

- 0 - the collision volume around the object is not recalculated.
- 1 - the collision volume is recalculated.

As we'll see in a later chapter, every 3D object is surrounded by an invisible box or sphere which is used to help determine when a collision between two objects has occurred. This is known as the **bounding volume**.

If, when we move to a different frame in an animated 3D object, that object has changed orientation, then we have the option to reposition the bounding volume to match the new position of the object (see FIG-36.9).

FIG-36.9 Recalculating the Bounding Volume for a New Frame



Failing to recalculate the bounding volume will give us inaccurate collision results.

Activity 36.12

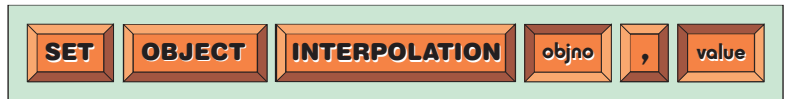
Modify your program so that when the hive alien stops, frame 13 is shown.

The SET OBJECT INTERPOLATION Statement

When an animation has to jump to a specific frame, this can create a rather jerky effect as the 3D object jumps suddenly from one position to another. By using the SET OBJECT INTERPOLATION statement, we can create a much smoother transition with the program actually creating temporary frames to move gradually from the current position to the new one. The format for this statement is shown in FIG-36.10.

FIG-36.10

The SET OBJECT INTERPOLATION Statement



In the diagram:

objno

is an integer value giving the ID of the 3D object to which this operation is to be applied.

value

is 1 to 100 and specifies the degree of interpolation to be used. A value of 100 gives instant transition to the final frame; a value of 1 will create a slow transition.

The program in LISTING-36.4 demonstrates the effect of the SET OBJECT INTERPOLATION statement by creating a jump from frame 0 to frame 13, first without interpolation, and, the second time, with interpolation.

LISTING-36.4

Smoother Frame
Movement using
Interpolation

```
REM *** Set screen resolution ***
SET DISPLAY MODE 1280, 1024, 32
COLOR BACKDROP 0
BACKDROP ON
AUTOCAM OFF
REM *** Create a point light ***
SET POINT LIGHT 0,-10,10,0
SET LIGHT RANGE 0,15
REM *** Load object; position camera ***
LOAD OBJECT "H-Alien Hivebrain-Move.X",1
POSITION CAMERA 0,1,-10
REM *** Jump to frame 13; no interpolation ***
REPEAT
    SET CURSOR 100,100
    PRINT "Press key for jump without interpolation"
UNTIL INKEY$() <> ""
SET OBJECT FRAME 1,13
WAIT 500
REM *** Jump to frame 13; no interpolation ***
REPEAT
    SET CURSOR 100,100
    PRINT "Press key to return to frame 0"
UNTIL INKEY$() <> ""
SET OBJECT FRAME 1,0
WAIT 500
REPEAT
    SET CURSOR 100,100
    PRINT "Press key for jump with interpolation"
UNTIL INKEY$() <> ""
SET OBJECT INTERPOLATION 1,1
SET OBJECT FRAME 1,13
REM *** End program ***
WAIT KEY
END
```

Activity 36.13

Type in and test the program in LISTING-36.4 (*loading04.dbpro*).

The APPEND OBJECT Statement

Often several animation files will exist which show the same object performing different actions. For example, the hive alien is shown moving into attack position in a file called *H-Alien Hivebrain-Attack1.x*. Rather than load this animation as a separate object, the frames the new file contains can be copied into the original object using the APPEND OBJECT statement which has the format shown in FIG-36.11.

FIG-36.11

The APPEND OBJECT
Statement



In the diagram:

filename

is a string giving the name of the file containing the 3D animation. This string may contain drive and path details.

objno

is an integer value giving the ID of the existing 3D object to which the new animation is to be added.

frame

is an integer value specifying the frame number at which the new animation is to be inserted into the existing object.

To add the alien attack animation to the end of the object containing the moving alien (object 1), we'd use the line:

```
APPEND OBJECT "H-Alien Hivebrain-Attack1.x", 1,  
↳ TOTAL OBJECT FRAMES (1) + 1
```

Activity 36.14

Modify program *loading03.dbpro* so that the alien attacks when it is at its closest point to the camera.

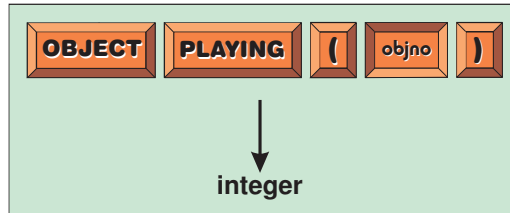
Retrieving Animation Object Information

The OBJECT PLAYING Statement

We can determine if an animated 3D object is currently playing using the OBJECT PLAYING statement which has the format shown in FIG-36.12.

FIG-36.12

The OBJECT PLAYING Statement



In the diagram:

objno

is an integer value giving the ID of the animated 3D object being checked.

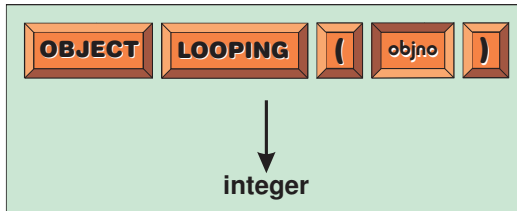
The statement returns 1 if the specified object is playing; otherwise zero is returned.

The OBJECT LOOPING Statement

To check if an animated 3D object is looping (rather than just playing) we can use the OBJECT LOOPING statement which has the format shown in FIG-36.13.

FIG-36.13

The OBJECT LOOPING Statement



In the diagram:

objno

is an integer value giving the ID of the animated 3D object being checked.

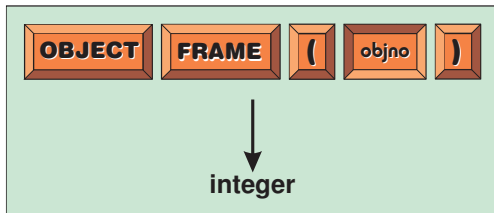
The statement returns 1 if the specified object is looping; otherwise zero is returned.

The OBJECT FRAME Statement

To discover which frame of an animated 3D object is currently being displayed, use the OBJECT FRAME statement which has the format shown in FIG-36.14.

FIG-36.14

The OBJECT FRAME
Statement



In the diagram:

objno

is an integer value giving the ID of the animated 3D object being checked.

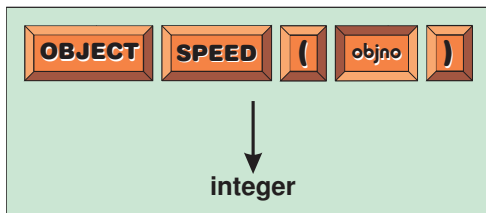
The function returns the number of the frame currently being displayed by the specified object.

The OBJECT SPEED Statement

The speed at which an animated 3D object is playing can be determined using the OBJECT SPEED statement whose format is shown in FIG-36.15.

FIG-36.15

The OBJECT SPEED
Statement



In the diagram:

objno

is an integer value giving the ID of the animated 3D object being checked.

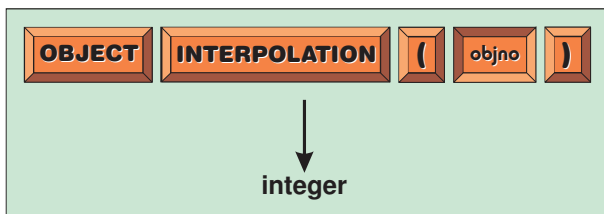
The statement returns the current speed setting. The default speed of a 3D animated object is 100, and that will be the value returned by this statement, unless the object in question has had its speed changed using the SET OBJECT SPEED.

The OBJECT INTERPOLATION Statement

The current interpolation value (given using SET OBJECT INTERPOLATION) can be determined using the OBJECT INTERPOLATION statement whose format is shown in FIG-36.16.

FIG-36.16

The OBJECT
INTERPOLATION
Statement



In the diagram:

objno

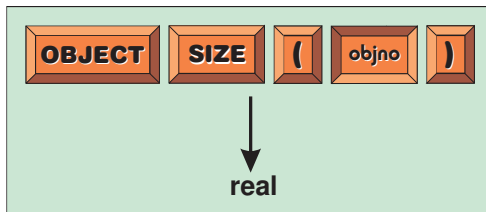
is an integer value giving the ID of the animated 3D object being checked.

The OBJECT SIZE Statement

The size of an object in each dimension can be discovered using the OBJECT SIZE statement. In fact, there are two separate versions of this statement. The first gives a single value representing the overall size of a 3D object, taking into account all three dimensions. The format for this statement is shown in FIG-36.17.

FIG-36.17

The OBJECT SIZE Statement (Version 1)



In the diagram:

objno

is an integer value specifying the ID of the object whose size is to be determined.

For example, we could display the overall size of object 1 using the line:

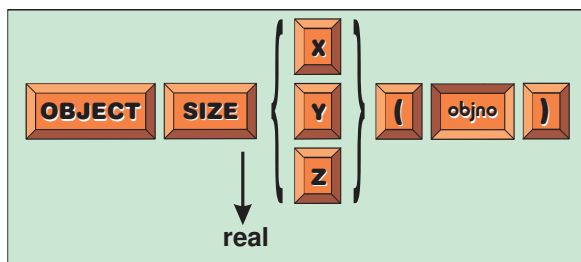
```
PRINT OBJECT SIZE (1)
```

The size is given in world units.

The second version of OBJECT SIZE is used to discover the actual size of the 3D object in each dimension. This version of the statement has the format shown in FIG-36.18.

FIG-36.18

The OBJECT SIZE Statement (Version 2)



In the diagram:

X,Y,Z

Choose the appropriate option for the dimension required (X = width; Y = height; Z = depth).

objno

is an integer value specifying the ID of the object whose size is to be determined.

For example, we could discover the height of object 1 using the line:

```
PRINT OBJECT SIZE Y (1)
```

The program in LISTING-36.5 loads the *H-Alien Hivebrain-Move.x* and displays the model's overall size as well as its width, height and depth.

LISTING-36.5

Finding the Size of a 3D Object

```
REM *** Set up screen ***
SET DISPLAY MODE 1280, 1024, 32
COLOR BACKDROP 0
BACKDROP ON
AUTOCAM OFF
POSITION CAMERA 0,0,-10
POINT CAMERA 0,0,0

REM *** Load object ***
LOAD OBJECT "H-Alien Hivebrain-Move.X",1
size# = OBJECT SIZE(1)
size#x = OBJECT SIZE X(1)
size#y = OBJECT SIZE Y(1)
size#z = OBJECT SIZE Z(1)
REPEAT
    SET CURSOR 100,100
    PRINT "Overall size: ",size#," width:",size#x,"
height:",size#y," depth:",size#z
UNTIL INKEY$() <> ""
WAIT 500
REM *** End program ***
END
```

Activity 36.15

Type in and test the program in LISTING-36.5 (*loading05.dbpro*).

Limbs

We looked at limbs in the previous chapter, but here we have a chance to examine the limbs of a model created outwith DarkBASIC Pro.

The program in LISTING-36.6 displays the names of all the limbs in the model *H-Alien Hivebrain-Move.x* and then hides one of the limbs.

LISTING-36.6

Examining the Limbs of an Imported Model

```
REM *** Set up screen ***
SET DISPLAY MODE 1280, 1024, 32
COLOR BACKDROP 0
BACKDROP ON
AUTOCAM OFF
POSITION CAMERA 0,0,-10
POINT CAMERA 0,0,0
SYNC ON

REM *** Load object ***
LOAD OBJECT "H-Alien Hivebrain-Move.X",1
SYNC

REM *** Get and display model limb details ***
PERFORM CHECKLIST FOR OBJECT LIMBS 1
nooflimbs = CHECKLIST QUANTITY()
PRINT "Model contains ",nooflimbs," limbs"
FOR c = 1 TO nooflimbs
    PRINT CHECKLIST STRING$(c)
NEXT c
SYNC

REM *** Return to automatic screen updating ***
WAIT KEY
SYNC OFF
REM *** End program ***
WAIT KEY
END
```


Activity 36.16

Type in and test the program in LISTING-36.6 (*loading06.dbpro*).

The limbs of an imported model can be manipulated using the statements covered in the previous chapter.

Activity 36.17

Modify your last program to hide limb 2 from the Hivebrain model.

Summary

- Complex 3D models are usually created using a 3D drawing package.
- DarkBASIC Pro can import .DBO, .3DS, .X and other format models.
- Models may be static or animated.
- Animated models consist of several frames - a bit like frames from a movie.
- Each frame of an animated model shows the model in a slightly different pose from the previous frame.
- Use `LOAD OBJECT` to load a model stored in a file.
- Make sure the model and the texturing it uses are stored in the current project folder.
- Use `PLAY OBJECT` to play a selected number of frames in an animated model.
- `PLAY OBJECT` plays through the specified frames only once.
- Use `LOOP OBJECT` to continuously play a set of frames from an animated model.
- Use `TOTAL OBJECT FRAMES` to discover exactly how many frames are used in an animated model.
- Use `SET OBJECT SPEED` to set the playing speed of a playing model animation.
- Use `STOP OBJECT` to halt the playing of an animated model.
- When an animation is stopped it continues to show the frame at which it halted.
- Use `SET OBJECT FRAME` to set an animated model to a specific frame.
- When setting the frame, the bounding volume may be recalculated.
- Use `SET OBJECT INTERPOLATION` to smooth the transition when jumping from one frame to another in an animated model.
- Use `APPEND OBJECT` to add the frames of one model to those of another.
- Use `OBJECT PLAYING` to determine if a specific animated model is playing.

- Use OBJECT LOOPING to determine if a specific animated model is currently looping.
- Use OBJECT FRAME to discover which frame is currently showing on an animated model.
- Use OBJECT SPEED to discover the current speed setting for an animated model.
- Use OBJECT INTERPOLATION to discover the interpolation setting for an animated model.
- The limbs of an imported model can be identified using the PERFORM CHECKLIST FOR OBJECT LIMBS statement covered in the previous chapter.
- The limbs of an imported model can be manipulated in the same way as limbs created using DarkBASIC Pro code.

Solutions

Activity 36.1

No solution required.

Activity 36.2

No solution required.

Activity 36.3

```
REM *** Set screen resolution ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP RGB(155,155,0)
BACKDROP ON
REM *** Load the jet object ***
LOAD OBJECT "H-Jet Fighter-Move.X",1
REM *** Position camera ***
POSITION CAMERA 0,0,-20
REM *** rotate jet ***
DO
    TURN OBJECT LEFT 1,1
    PITCH OBJECT UP 1,1
    WAIT 10
LOOP
END
```

Activity 36.4

No solution required.

Activity 36.5

```
REM *** Set screen resolution ***
SET DISPLAY MODE 1280, 1024, 32
COLOR BACKDROP RGB(155,155,0)
BACKDROP ON
REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 0,1,-10
REM *** Create a point light ***
SET POINT LIGHT 0,-10,10,0
SET LIGHT RANGE 0,15
REM *** Load object ***
LOAD OBJECT "H-Alien Hivebrain-Move.X",1
WAIT KEY
PLAY OBJECT 1
REM *** End program ***
WAIT KEY
END
```

Activity 36.6

```
REM *** Set screen resolution ***
SET DISPLAY MODE 1280, 1024, 32
COLOR BACKDROP RGB(155,155,0)
BACKDROP ON
REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 0,1,-10
REM *** Create a point light ***
SET POINT LIGHT 0,-10,10,0
SET LIGHT RANGE 0,15
REM *** Load object ***
LOAD OBJECT "H-Alien Hivebrain-Move.X",1
WAIT KEY
LOOP OBJECT 1
REM *** End program ***
WAIT KEY
END
```

Activity 36.7

```
REM *** Set screen resolution ***
SET DISPLAY MODE 1280, 1024, 32
COLOR BACKDROP RGB(155,155,0)
BACKDROP ON
REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 0,1,-10
REM *** Create a point light ***
SET POINT LIGHT 0,-10,10,0
SET LIGHT RANGE 0,15
REM *** Load object ***
LOAD OBJECT "H-Alien Hivebrain-Move.X",1
WAIT KEY
LOOP OBJECT 1
REM *** Display number of frames ***
totalframes = TOTAL OBJECT FRAMES(1)
DO
    SET CURSOR 100,100
    PRINT "Total frames = ",totalframes
LOOP
REM *** End program ***
WAIT KEY
END
```

Activity 36.8

```
REM *** Set screen resolution ***
SET DISPLAY MODE 1280, 1024, 32
COLOR BACKDROP RGB(155,155,0)
BACKDROP ON
REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 0,1,-10
REM *** Create a point light ***
SET POINT LIGHT 0,-10,10,0
SET LIGHT RANGE 0,15
REM *** Load object ***
LOAD OBJECT "H-Alien Hivebrain-Move.X",1
WAIT KEY
LOOP OBJECT 1,0,20
totalframes = TOTAL OBJECT FRAMES(1)
DO
    SET CURSOR 100,100
    PRINT "Total frames = ",totalframes
LOOP
REM *** End program ***
WAIT KEY
END
```

Activity 36.9

No solution required.

Activity 36.10

```
REM *** Set screen resolution ***
SET DISPLAY MODE 1280, 1024, 32
COLOR BACKDROP 0
BACKDROP ON
AUTOCAM OFF
REM *** Create a point light ***
SET POINT LIGHT 0,-10,10,0
SET LIGHT RANGE 0,15
REM *** Load object; position camera ***
LOAD OBJECT "H-Alien Hivebrain-Move.X",1
POSITION CAMERA 0,1,-10
REM *** Play object continuously ***
LOOP OBJECT 1,0,20
REM *** Change play speed ***
SET OBJECT SPEED 1,50
```

```

REM *** Move alien ***
DO
    MOVE OBJECT 1,-0.03
LOOP
REM *** End program ***
END

```

Activity 36.11

```

REM *** Set screen resolution ***
SET DISPLAY MODE 1280, 1024, 32
COLOR BACKDROP 0
BACKDROP ON
AUTOCAM OFF
REM *** Create a point light ***
SET POINT LIGHT 0,-10,10,0
SET LIGHT RANGE 0,15
REM *** Load object; position camera ***
LOAD OBJECT "H-Alien Hivebrain-Move.X",1
POSITION CAMERA 0,1,-10
REM *** Play object continuously ***
LOOP OBJECT 1,0,20
REM *** Change play speed ***
SET OBJECT SPEED 1,50
REM *** Move alien ***
DO
    MOVE OBJECT 1,-0.03
    REM *** Stop alien when its close ***
    IF OBJECT POSITION Z(1) < -6
        STOP OBJECT 1
        EXIT
    ENDIF
LOOP
REM *** End program ***
WAIT KEY
END

```

Activity 36.12

```

REM *** Set screen resolution ***
SET DISPLAY MODE 1280, 1024, 32
COLOR BACKDROP 0
BACKDROP ON
AUTOCAM OFF
REM *** Create a point light ***
SET POINT LIGHT 0,-10,10,0
SET LIGHT RANGE 0,15
REM *** Load object; position camera ***
LOAD OBJECT "H-Alien Hivebrain-Move.X",1
POSITION CAMERA 0,1,-10
REM *** Play object continuously ***
LOOP OBJECT 1,0,20
REM *** Change play speed ***
SET OBJECT SPEED 1,50
REM *** Move alien ***
DO
    MOVE OBJECT 1,-0.03
    REM *** Stop alien when its close ***
    IF OBJECT POSITION Z(1) < -6
        STOP OBJECT 1
        SET OBJECT FRAME 1,13
        EXIT
    ENDIF
LOOP
REM *** End program ***
WAIT KEY
END

```

Activity 36.13

No solution required.

Activity 36.14

```

REM *** Set screen resolution ***
SET DISPLAY MODE 1280, 1024, 32
COLOR BACKDROP 0
BACKDROP ON
AUTOCAM OFF
REM *** Create a point light ***
SET POINT LIGHT 0,-10,10,0
SET LIGHT RANGE 0,15
REM *** Load object ***
LOAD OBJECT "H-Alien Hivebrain-Move.X",1
REM *** Add attack frames ***
APPEND OBJECT "H-Alien
Hivebrain-Attack1.x",1,26
REM *** Position camera ***
POSITION CAMERA 0,1,-10
REM *** Play object continuously ***
LOOP OBJECT 1,0,20
REM *** Change play speed ***
SET OBJECT SPEED 1,50
REM *** Move alien ***
DO
    MOVE OBJECT 1,-0.03
    REM *** Stop alien when its close ***
    IF OBJECT POSITION Z(1) < -6
        STOP OBJECT 1
        PLAY OBJECT 1,26,
        ↵TOTAL OBJECT FRAMES(1)
        EXIT
    ENDIF
LOOP
REM *** End program ***
WAIT KEY
END

```

Activity 36.15

No solution required.

Activity 36.16

No solution required.

Activity 36.17

To hide limb 2 add the lines

```

REM *** Hide a limb ***
HIDE LIMB 1,2

```

just before the line

```

REM *** End program ***

```

Screen Control

How to Check if an Object is on Screen

How to Select an Object on the Screen

Calculating Distance from the Camera

Calculating Position Relative to the Camera

Screen Coordinates and World Coordinates

Using the Mouse for Selection

Introduction

There must be very few games in which the user does not directly control the movement of at least one 3D object. For example, if we created a 3D chess game, we'd need to allow the player to select which piece is to be moved next.

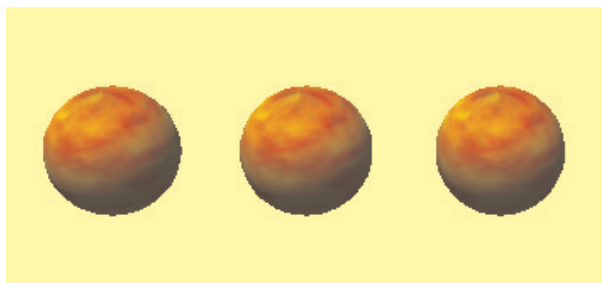
In this chapter we're going to look at some techniques for helping the player to select from a group of 3D objects, as well as finding out the commands available to help with this task.

Selecting an Object

FIG-37.1 shows a line of three balls. The aim of this section is to examine ways of selecting one of the three balls for deletion.

FIG-37.1

A Choice of Objects



The code required to create this set up is given in LISTING-37.1.

LISTING-37.1

Creating the Objects

```
FUNCTION ScreenSetUp()  
  REM *** Set up screen ***  
  SET DISPLAY MODE 1280,1024,32  
  COLOR BACKDROP 0  
  BACKDROP ON  
  REM *** Position camera ***  
  AUTOCAM OFF  
  POSITION CAMERA 0,10,-20  
  POINT CAMERA 0,0,0  
ENDFUNCTION  
  
FUNCTION CreateBalls()  
  #CONSTANT firstballobj 11  
  #CONSTANT lastballobj 13  
  #CONSTANT ballimg 2  
  
  REM *** Load ball texture ***  
  LOAD IMAGE "lava1.bmp",ballimg  
  
  REM *** Create balls ***  
  x = -9  
  FOR objno = firstballobj TO lastballobj  
    MAKE OBJECT SPHERE objno,2,20,20  
    TEXTURE OBJECT objno,ballimg  
    POSITION OBJECT objno,x,0,0  
    x = x + 3  
  NEXT objno  
ENDFUNCTION
```

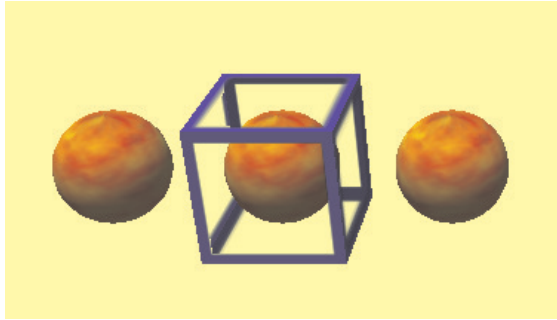
Activity 37.1

Type in the code given above (*control01.dbpro*) and add a main section to call these functions and display the balls on the screen.

Next, we will add a selector. This would give the player a visual clue as to which object is currently selected (see FIG-37.2).

FIG-37.2

Adding a Selector



The new object is simply a cube textured with a blue square and a black centre (see FIG-37.3). The black area of the object has been made transparent.

FIG-37.3

The Texture Used on the Selector Cube



A function to add the selector is given below:

```
FUNCTION CreateSelector()  
  #CONSTANT selectorobj 2  
  #CONSTANT selectorimg 3  
  
  REM *** Load selector texture ***  
  LOAD IMAGE "selector.bmp",selectorimg  
  
  REM *** Create selector cube ***  
  MAKE OBJECT CUBE selectorobj,2  
  TEXTURE OBJECT selectorobj, selectorimg  
  SET OBJECT TRANSPARENCY selectorobj,1  
  POSITION OBJECT selectorobj,0,0,0  
  SET OBJECT CULL selectorobj,0  
ENDFUNCTION
```

Activity 37.2

Add the new function to your program (calling it from the main section) and check that the selector is created successfully.

The next function allows the user to move the selector left and right using the arrow keys and to make a final selection by pressing the *Enter* key.

```
FUNCTION MoveSelector()  
  REPEAT  
    REM *** Get current selector position ***  
    x = OBJECT POSITION X(selectorobj)  
    y = OBJECT POSITION Y(selectorobj)
```

```

z = OBJECT POSITION Z(selectorobj)
REM *** Move selector left or right ***
IF LEFTKEY()
    POSITION OBJECT selectorobj,x-3,y,z
ELSE IF RIGHTKEY()
    POSITION OBJECT selectorobj,x+3,0,0
ENDIF
ENDIF
WAIT 100
UNTIL INKEY$() = CHR$(13)
ENDFUNCTION

```

CHR\$(13) is the Enter (or Return) key.

The WAIT 100 statement at the end of the function is there to slow down the whole process. If not included the selector is too responsive to key presses.

Activity 37.3

Add the new function to your program and test that it works correctly.

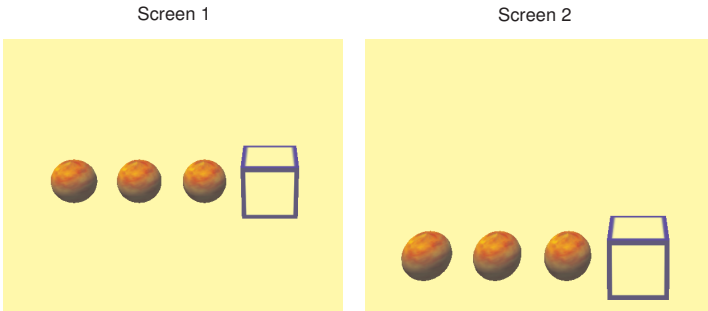
When the selector has moved over the required ball, we can see on the screen which item is selected, but we also need to get the program itself to realise which ball has been picked. To do that we need to learn a few more DarkBASIC Pro statements.

The OBJECT SCREEN Statement

3D objects exist in a 3 dimensional space and yet are displayed on a two-dimensional screen. Although an object may remain at a fixed point in 3D space, still it can move position on the screen if the camera were to move (see FIG-37.4).

FIG-37.4

Screen Position Changes though 3D Position is Unchanged

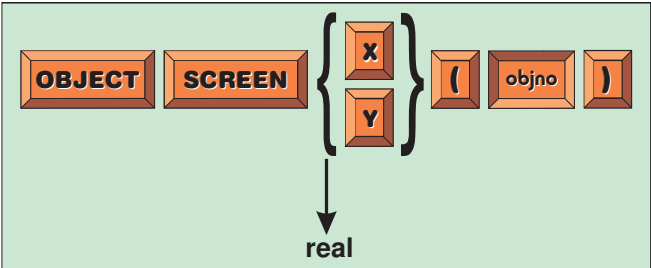


The objects shift to a different position on the screen as the camera is moved but remain fixed at the same location in world space.

We can use the OBJECT SCREEN statement to discover the x,y coordinates of a 3D object on the screen. The statement has the format shown in FIG-37.5

FIG-37.5

The OBJECT SCREEN Statement



In the diagram:

X,Y

Use one of these to select the required ordinate.

objno

is an integer value specifying the object whose screen position is to be determined.

For example, we could discover the screen coordinates of the selector object in the program we are developing using the lines:

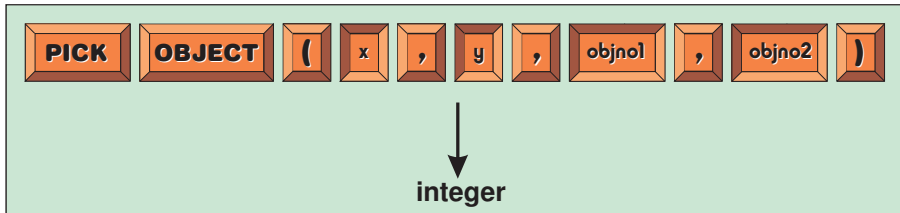
```
selectorx# = OBJECT SCREEN X(selectorobj)
selectory# = OBJECT SCREEN Y(selectorobj)
```

The PICK OBJECT Statement

The PICK OBJECT statement complements the OBJECT SCREEN statement by returning the ID of the object which lies at a particular spot on the screen. The statement searches through all objects within a given set of object IDs and returns the ID of the first object it finds at the screen coordinates given. The PICK OBJECT statement has the format shown in FIG-37.6.

FIG-37.6

The PICK OBJECT Statement



In the diagram:

x,y are a pair of integer values giving the screen coordinates to be checked.

objno1 is an integer value giving the lowest object ID to be checked.

objno2 is an integer value giving the highest object ID to be checked.

If any part of an object with an ID between *objno1* and *objno2* is positioned at screen coordinates (*x,y*), then the ID of that object is returned by the statement. If point (*x,y*) does not lie within any of the objects, zero is returned.

For example, we could check which ball covers the screen position (*selectorx#*, *selectory#*) with the line:

```
objno =
PICK OBJECT(selectorx#,selectory#,firstballobj,lastballobj)
```

Now we have enough information to create a new function, *GetObjectSelected()*, which returns the ID of the ball selected.

```
FUNCTION GetObjectSelected()
  selectorx# = OBJECT SCREEN X(selectorobj)
  selectory# = OBJECT SCREEN Y(selectorobj)
  objno =
  PICK OBJECT(selectorx#,selectory#,firstballobj,lastballobj)
ENDFUNCTION objno
```

To finish off, we'll create a routine to handle the ball selected. For the moment we'll just make the ball disappear.

```

FUNCTION HandleObject(objno)
    REM *** IF no object selected, exit ***
    IF objno = 0
        EXITFUNCTION
    ENDIF
    REM *** Hide object selected ***
    HIDE OBJECT objno
ENDFUNCTION

```

Activity 37.4

Add the functions *GetObjectSelected()* and *HandleObject()*, to your program.

Code the main section of the program as follows:

```

ScreenSetUp()
CreateBalls()
CreateSelector()
DO
    MoveSelector()
    objno = GetObjectSelected()
    HandleObject(objno)
LOOP
WAIT KEY
END

```

Test the program by hiding all three balls.

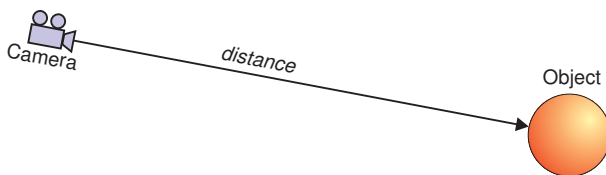
When a PICK OBJECT statement is executed, it generates several items of data. These are the distance from the camera to the object selected and the position of that object in 3D space relative to the camera. The following two statements allow us to access this data.

The GET PICK DISTANCE Statement

Executing the PICK OBJECT statement causes DarkBASIC Pro to calculate the distance between the camera and the object selected (see FIG-37.7).

FIG-37.7

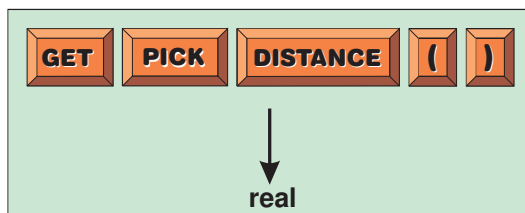
The Camera-Object
Distance



We can access this information using the GET PICK DISTANCE statement which has the format shown in FIG-37.8.

FIG-37.8

The GET PICK
DISTANCE Statement



The statement returns the distance in world units between the camera and the surface of the object returned by the previous PICK OBJECT statement.

This information would be useful if we needed to calculate if an enemy is within hitting distance in a First Person Shooter game.

We could rewrite the *HandleObject()* routine to display the distance of the selected sphere:

```

FUNCTION HandleObject(objno)
  REM *** IF no object selected, exit ***
  IF objno = 0
    EXITFUNCTION
  ENDIF
  REM *** Find distance from camera to ball ***
  distance# = GET PICK DISTANCE()
  INK 0,0
  REM *** Display distance for 2 seconds ***
  t = TIMER()
  WHILE TIMER() - t < 2000
    SET CURSOR 100,100
    PRINT "Distance = ",distance#
  ENDWHILE
ENDFUNCTION

```

Activity 37.5

Modify your *HandleObject()* function to match the code given above.

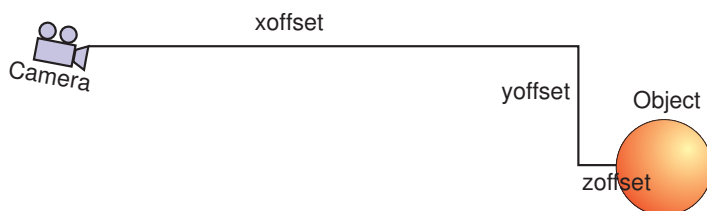
Test the routine by checking the distance of all three balls from the camera.

The PICK VECTOR Statement

The second data item created by executing the PICK OBJECT statement is a 3D vector containing the coordinates of the selected object. The coordinates are not the world coordinates of the object (which we could obtain using OBJECT POSITION) but the position of the object as measured from the camera (see FIG-37.9).

FIG-37.9

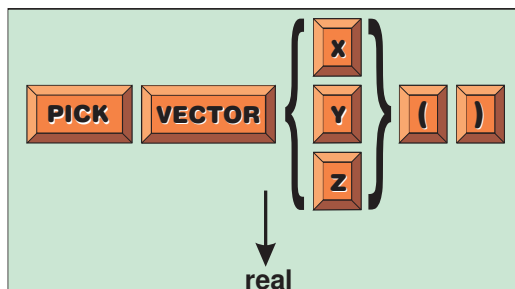
PICK VECTOR gives the Position of the Selected Object relative to the Camera



We can examine the contents of this vector using the PICK VECTOR statement which is shown in FIG-37.10.

FIG-37.10

The PICK VECTOR Statement



In the diagram:

X,Y,Z

Use the appropriate option for the ordinate required.

For example, if we assume the statement

```
objno = PICK OBJECT (640,512,0,2)
```

returns a non-zero value, we could get the returned object's position (relative to the camera) using the lines:

```
x# = GET PICK VECTOR X()  
y# = GET PICK VECTOR Y()  
z# = GET PICK VECTOR Z()
```

The program in LISTING-37.2 allows the user to move the camera using the arrow keys and displays a continual readout of the relative position of a sphere in relation to the camera.

LISTING-37.2

An Object's
Coordinates Relative to
the Camera

```
REM *** Set up screen ***  
SET DISPLAY MODE 1280,1024,32  
AUTOCAM OFF  
POSITION CAMERA 0,0,-10  
POINT CAMERA 0,0,0  
  
REM *** Create sphere ***  
MAKE OBJECT SPHERE 1,10  
POSITION OBJECT 1,0,0,20  
  
REM *** main loop ***  
DO  
    objno = PICK OBJECT (640,512,0,2)  
    DisplayCoords(objno)  
    CONTROL CAMERA USING ARROWKEYS 0,0.1,0.1  
LOOP  
END  
  
FUNCTION DisplayCoords(objno)  
    REM *** Get object's relative coords ***  
    x# = GET PICK VECTOR X()  
    y# = GET PICK VECTOR Y()  
    z# = GET PICK VECTOR Z()  
    REM *** Display info on screen ***  
    SET CURSOR 100,100  
    PRINT "Object:",objno,"Relative coords   x:",x#," Y:",y#,  
        " z:",z#  
ENDFUNCTION
```

Activity 37.6

Type in and test the program given in LISTING-37.2 (*control02.dbpro*).

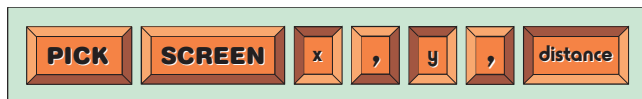
The PICK SCREEN Statement

It is possible to pick an arbitrary point on the screen, an arbitrary distance "into" the screen and have DarkBASIC Pro calculate the coordinates of that point in 3D space relative to the camera. This information is stored in the PICK vector.

The operation is performed using the PICK SCREEN statement which has the format shown in FIG-37.11.

FIG-37.11

The PICK SCREEN
Statement



In the diagram:

x,y

are a pair of real values giving the screen coordinates chosen.

distance

is a real value giving the depth factor to be taken into account.

For example, if we were to execute the line

```
PICK SCREEN 200,300,15
```

the computer would work out the equivalent point in the 3D world and return the coordinates of that point relative to the active camera's position.

The program in LISTING-37.3 allows the user to mouse click anywhere on the screen and returns the coordinates of the equivalent point at a screen depth of 15 units.

LISTING-37.3

Screen Coordinates
Converted to World
Coordinates

```
REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
AUTOCAM OFF
POSITION CAMERA 0,0,-10
POINT CAMERA 0,0,0
REM *** Create object to create correct screen mode ***
MAKE OBJECT SPHERE 1,0
REM *** main loop ***
DO
    DisplayCoords()
LOOP
END

FUNCTION DisplayCoords()
    REM *** Get mouse coords ***
    PICK SCREEN MOUSEX(), MOUSEY(), 15
    x# = GET PICK VECTOR X()
    y# = GET PICK VECTOR Y()
    z# = GET PICK VECTOR Z()
    REM *** Display info on screen ***
    SET CURSOR 100,100
    PRINT "Relative coords   x:",x#," Y:",y#," z:",z#
ENDFUNCTION
```

Activity 37.7

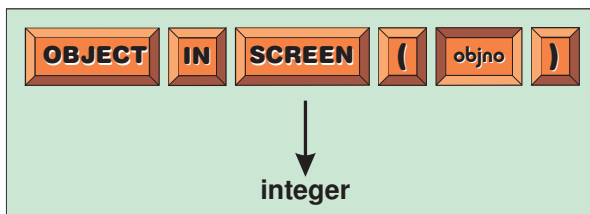
Type in and test the program in LISTING-37.3 (*control03.dbpro*). How could we convert the relative coordinates to absolute coordinates?

The OBJECT IN SCREEN Statement

We can check to see if a specific object actually appears within the screen area using the OBJECT IN SCREEN statement which has the format shown in FIG-37.12.

FIG-37.12

The OBJECT IN
SCREEN Statement



In the diagram:

objno

is an integer value giving the ID of the object to be checked.

The routine returns 1 if the object specified is wholly or partially on the screen; otherwise zero is returned.

The program in LISTING-37.4 displays the on-screen/off-screen status of a sphere as it moves slowly towards the edge of the screen.

LISTING-37.4

Checking if an Object is
On Screen

```
REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
AUTOCAM OFF
POSITION CAMERA 0,0,-20
POINT CAMERA 0,0,0
MAKE OBJECT SPHERE 1,5
POINT OBJECT 1,1,0,0

REM *** main loop ***
DO
    MOVE OBJECT 1,0.01
    DisplayStatus(1)
LOOP
END

FUNCTION DisplayStatus(objno)
    REM *** Display info on screen ***
    SET CURSOR 100,100
    IF OBJECT IN SCREEN(objno)
        PRINT "On screen"
    ELSE
        PRINT "Off screen"
    ENDIF
ENDFUNCTION
```

Activity 37.8

Type in and test the program in LISTING-37.4 (*control04.dbpro*).

Are the same results obtained when the object is invisible?

What happens when an object passes behind the camera?

Selecting Objects using the Mouse

We can use the mouse to select an object on the screen by moving the mouse pointer over the required object and then feeding the mouse coordinates as parameters to the PICK OBJECT statement.

The program in LISTING-37.5 is a variation on the three-ball example we created earlier in this chapter. By clicking on a ball with the mouse, the ball will disappear.

LISTING-37.5

Selecting an Object
Using the Mouse

```
ScreenSetUp()
CreateBalls()
DO
    objno = GetObjectSelected()
    HandleObject(objno)
LOOP
```

continued on next page

LISTING-37.5

(continued)

Selecting an Object
Using the Mouse

```
REM *** End program ***
END

FUNCTION ScreenSetUp()
    REM *** Set up screen ***
    SET DISPLAY MODE 1280, 1024, 32
    COLOR BACKDROP RGB(255,255,150)
    BACKDROP ON
    REM *** Position camera ***
    AUTOCAM OFF
    POSITION CAMERA 0,10,-20
    POINT CAMERA 0,0,0
ENDFUNCTION

FUNCTION CreateBalls()
    #CONSTANT firstballobj 11
    #CONSTANT lastballobj 13
    #CONSTANT ballimg 2
    REM *** Load ball texture ***
    LOAD IMAGE "lava1.bmp",ballimg
    REM *** Create balls ***
    x = -9
    FOR objno = firstballobj TO lastballobj
        MAKE OBJECT SPHERE objno,2,20,20
        TEXTURE OBJECT objno,ballimg
        POSITION OBJECT objno,x,0,0
        x = x + 3
    NEXT objno
ENDFUNCTION

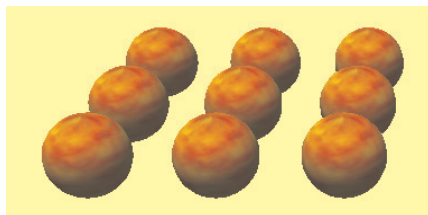
FUNCTION GetObjectSelected()
    REM *** Wait for mouse click ***
    WHILE MOUSECLICK()=0
    ENDWHILE
    REM *** Select object under mouse pointer ***
    objno = PICK OBJECT(MOUSEX(),MOUSEY(),firstballobj,lastballobj)
ENDFUNCTION objno

FUNCTION HandleObject(objno)
    REM *** IF no object selected, exit ***
    IF objno = 0
        EXITFUNCTION
    ENDIF
    HIDE OBJECT objno
ENDFUNCTION
```

Activity 37.9

Type in and test the program in LISTING-37.5 (*control05.dbpro*).

Modify the program to have a 3 by 3 grid of balls. Each new row of balls should be displaced by 3 units along the z-axis (see below).



Check that each of the nine balls can be deleted.

Summary

- DarkBASIC Pro has several statements that allow us to relate 2D screen coordinates to the world coordinates used by 3D objects.
- Use the OBJECT SCREEN statement to discover the screen coordinates of a specific 3D object.
- Use the PICK OBJECT statement to discover the ID of an object at a given point on the screen.
- Use GET PICK DISTANCE to discover how far the object detected by PICK OBJECT is from the camera.
- Use PICK VECTOR to discover the coordinates of the object detected by PICK OBJECT relative to the camera.
- Use PICK SCREEN to convert a 2D screen position and a depth value to a point in 3D space. The coordinates generated are relative to the camera position.
- Use OBJECT IN SCREEN to determine if a specific object is within the view of the camera (and hence, appearing on the screen).

Solutions

Activity 37.1

The main section should be coded as:

```
REM *** Main section ***
ScreenSetUp()
CreateBalls()
REM *** End program ***
WAIT KEY
END
```

Activity 37.2

The main section is now:

```
REM *** Main section ***
ScreenSetUp()
CreateBalls()
CreateSelector()
REM *** End program ***
WAIT KEY
END
```

Activity 37.3

The main section changes to:

```
REM *** Main section ***
ScreenSetUp()
CreateBalls()
CreateSelector()
MoveSelector()
REM *** End program ***
WAIT KEY
END
```

Activity 37.4

The complete version of the program is:

```
REM *** Main section ***
ScreenSetUp()
CreateBalls()
CreateSelector()
DO
    MoveSelector()
    objno = GetObjectSelected()
    HandleObject(objno)
LOOP
REM *** End program ***
END
```

```
FUNCTION ScreenSetUp()
    REM *** Set up screen ***
    SET DISPLAY MODE 1280,1024,32
    COLOR BACKDROP RGB(255,255,100)
    BACKDROP ON
    REM *** Position camera ***
    AUTOCAM OFF
    POSITION CAMERA 0,10,-20
    POINT CAMERA 0,0,0
ENDFUNCTION
```

```
FUNCTION CreateBalls()
    #CONSTANT firstballobj 11
    #CONSTANT lastballobj 13
    #CONSTANT ballimg 2
    REM *** Load ball texture ***
    LOAD IMAGE "laval.bmp",ballimg
    REM *** Create balls ***
```

```
x = -9
```

```
FOR objno = firstballobj TO lastballobj
    MAKE OBJECT SPHERE objno,2,20,20
    TEXTURE OBJECT objno,ballimg
    POSITION OBJECT objno,x,0,0
    x = x + 3
NEXT objno
ENDFUNCTION
```

```
FUNCTION CreateSelector()
    #CONSTANT selectorobj 2
    #CONSTANT selectorimg 3
    REM *** Load selector texture ***
    LOAD IMAGE "selector.bmp",selectorimg
    REM *** Create selector cube ***
    MAKE OBJECT CUBE selectorobj,2
    TEXTURE OBJECT selectorobj, selectorimg
    SET OBJECT TRANSPARENCY selectorobj,1
    POSITION OBJECT selectorobj,0,0,0
    SET OBJECT CULL selectorobj,0
ENDFUNCTION
```

```
FUNCTION GetObjectSelected()
    selectorx# = OBJECT SCREEN X(selectorobj)
    selectory# = OBJECT SCREEN Y(selectorobj)
    objno = PICK OBJECT
    (selectorx#,selectory#,firstballobj,
    lastballobj)
ENDFUNCTION objno
```

```
FUNCTION HandleObject(objno)
    REM *** IF no object selected, exit ***
    IF objno = 0
        EXITFUNCTION
    ENDIF
    REM *** Hide object selected ***
    HIDE OBJECT objno
ENDFUNCTION
```

```
FUNCTION MoveSelector()
    REPEAT
        REM *** Get selector position ***
        x = OBJECT POSITION X(selectorobj)
        y = OBJECT POSITION Y(selectorobj)
        z = OBJECT POSITION Z(selectorobj)
        REM *** Move selector left/right ***
        IF LEFTKEY()
            POSITION OBJECT selectorobj,x-3,y,z
        ELSE IF RIGHTKEY()
            POSITION OBJECT selectorobj,x+3,0,0
        ENDIF
        ENDIF
        WAIT 100
    UNTIL INKEY$() = CHR$(13)
ENDFUNCTION
```

Activity 37.5

No solution required.

Activity 37.6

No solution required.

Activity 37.7

Adding the camera's coordinates to the relative coordinates gives us the absolute coordinates.

Activity 37.8

Being hidden makes no difference to an object's on-screen status.

We could make the object move behind the camera by moving it along the z-axis. This only requires the direction in which the sphere is pointing to be changed:

```
POINT OBJECT 1,0,0,-1
```

The object is considered to be off-screen when it passes behind the camera.

Activity 37.9

To create nine balls we need to modify the *CreateBalls()* function to create 3 rows of 3. The new version of the routine given below keeps changes to a minimum.

```
FUNCTION CreateBalls()  
  #CONSTANT firstballobj  11  
  #CONSTANT lastballobj   19  
  #CONSTANT ballimg  2  
  REM *** Load ball texture ***  
  LOAD IMAGE "laval.bmp",ballimg  
  REM *** Create balls ***  
  x = -9  
  z = 0  
  FOR objno = firstballobj TO lastballobj  
    MAKE OBJECT SPHERE objno,2,20,20  
    TEXTURE OBJECT objno,ballimg  
    POSITION OBJECT objno,x,0,z  
    x = x + 3  
    IF x = 0  
      x = -9  
      z = z + 3  
    ENDIF  
  NEXT objno  
ENDFUNCTION
```

Adding Help to a Game

Constructing a 3D Game of Skill

Recording Game State

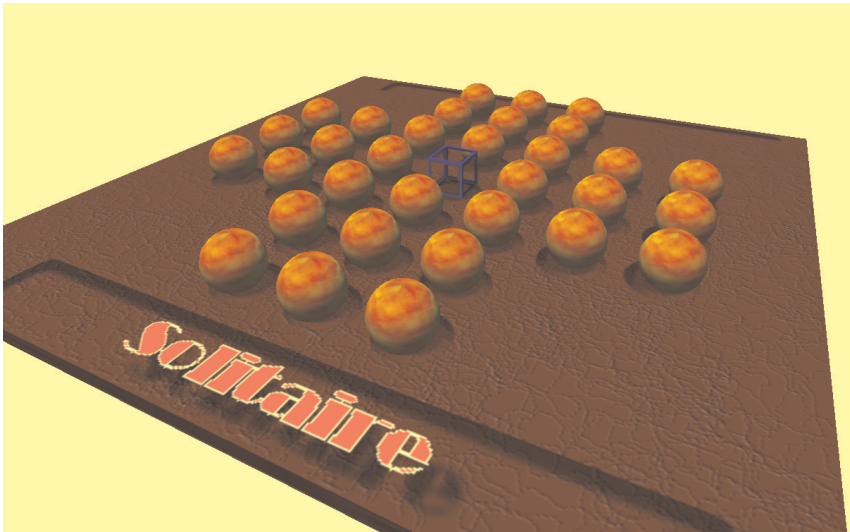
Solitaire - The Board Game

Introduction

In this chapter we are going to develop a 3D version of the board game Solitaire (see FIG-38.1) - not to be confused with the card game of the same name.

FIG-38.1

The Game of Solitaire



The Equipment

The game consists of 32 marbles and a board with 33 indentations, or pits. The pits form the shape of a plus sign (+). A marble is placed in every pit, except the central one.

The Aim

The aim of the game is to remove all but one marble from the board. This last marble should be in the central pit at the end of the game.

The Rules

Any marble can be moved by jumping over one other marble into an empty pit. Jumps can be horizontal or vertical but not diagonal. Only a single marble can be jumped over on each move. The marble which is jumped over is removed from the board.

Creating a Computer Version of the Game

User Controls

Each turn involves the player first selecting a marble and then an empty pit into which the marble is to be moved.

To achieve this, the player will move a selector cube about the board using the arrow keys. Once over a required pit, the Enter key is pressed to indicate a selection.

Pressing U will allow the player to deselect a marble.

Pressing N will initiate a new game.

Pressing F will terminate the current game.

Pressing F1 will display the help screen. Within the help screen, pressing R will display the rules, and K the keys which may be used in the game.

Game Responses

The game will respond to player actions in the following ways:

- When a marble is selected, it will change colour.
- When an empty pit is selected as the second part of a move, the selected marble will jump to the new position and return to its original colour. The jumped marble will be removed.
- When U is pressed immediately after a marble has been selected, the selected marble will return to its original colour and the player will have to select a new marble.
- If an empty pit is selected when a marble selection is required, an error message will be displayed and the player must select a pit containing a marble.
- If an occupied pit is selected when a empty pit is required, an error message will be displayed and a new destination pit must be selected.
- If the marble and destination pit selected do not generate a valid move, an error message will be displayed and a new destination pit must be selected.
- If N is pressed, a new game is started. No prompt is given.
- If F is pressed, the program will terminate after displaying a shut down message.
- If F1 is pressed, the introductory page of the Help section will be displayed. A second press of the same key will cause the Help section to disappear.
- If R is pressed while a help page is visible, the help page containing the rules will be displayed.
- IF K is pressed while a help page is visible, the help page showing the keys which can be used in the game will be displayed.

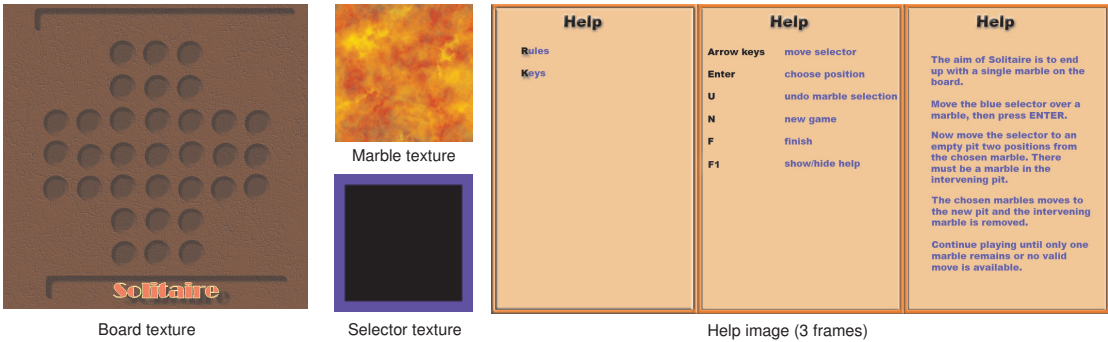
Screen Layout

The screen layout is shown in FIG-38.1.

Media Used

The images used to texture the board, marbles and selector cube are shown in FIG-38.2. Notice that even the help text is held as a single image with three frames.

FIG-38.2 The Images Used in the Program



Data Structures

Although we can use 3D objects to represent the board visually, we need a second way of representing the board as data. This will make the coding easier when trying to decide on valid moves.

If we had a square playing area, the obvious approach would be to use a two-dimensional array with one cell representing one pit on the board. However, despite using a cross-shaped playing area, with a little thought we can still make use of a two-dimensional array. At the centre the board is 7 pits wide and 7 pits deep, so a 7 by 7 integer array could be used to represent the board:

```
DIM board (7,7)
```

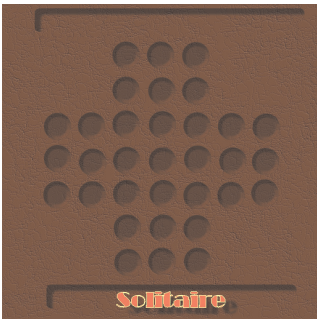
Now we'll fill the array with a zero in any cell that corresponds to a pit and with -1 where a cell corresponds to a flat part of the board. FIG-38.3 shows the board and the corresponding array contents.

FIG-38.3

The Board and the Array used to Represent it

-1	-1	0	0	0	-1	-1
-1	-1	0	0	0	-1	-1
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
-1	-1	0	0	0	-1	-1
-1	-1	0	0	0	-1	-1

board Array



Actual Board Layout

Lower ID values are reserved for other objects.

Rather than just store zeros, it will suit us better to store the IDs of each marble in the cells of the array. So, if we assume the marbles have object IDs starting at 11, the board array would start off as shown in FIG-38.4.

FIG-38.4

The *board* array with Marble IDs

-1	-1	11	12	13	-1	-1
-1	-1	14	15	16	-1	-1
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31	32	33	34	35	36	37
-1	-1	38	39	40	-1	-1
-1	-1	41	42	43	-1	-1

board Array

It would be useful to employ a second 7 by 7 array in which to store the world coordinates of each pit. Although we could work the positions out mathematically (since the pits are evenly spaced) it will probably create less complex code if we just store the coordinates of each position in another array. Since the board is flat, it won't be necessary to store the y value of any position (since these will all be identical), just the x and z values.

When we create a 3D object, its centre is at (0,0,0), so, assuming we don't move the board, the centre pit will have x, z values of 0,0.

Since we need to store two values in every cell of the array (the x and z ordinates), we'll need to create a record structure for this

```

TYPE Coords
  x AS INTEGER
  z AS INTEGER
ENDTYPE

```

before creating our array:

```

DIM boardCoords(7,7) AS Coords

```

Now, assuming the centre of each pit is 3 world units in both directions from any neighbour, the new array would contain the values shown in FIG-38.5.

FIG-38.5

The *boardCoords* Array

-1	-1	(-3,9)	(0,9)	(3,9)	-1	-1
-1	-1	(-3,6)	(0,6)	(3,6)	-1	-1
(-9,3)	(-6,3)	(-3,3)	(0,3)	(3,3)	(6,3)	(9,3)
(-9,0)	(-6,0)	(-3,0)	(0,0)	(3,0)	(6,0)	(9,0)
(-9,-3)	(-6,-3)	(-3,-3)	(0,-3)	(3,-3)	(6,-3)	(9,-3)
-1	-1	(-3,-6)	(0,-6)	(3,-6)	-1	-1
-1	-1	(-3,-9)	(0,-9)	(3,-9)	-1	-1

***boardCoords* Array**

In practice, we'll probably store coordinates in the invalid positions of the array (the four cells in each corner), since it is easier to write the code that way.

We'll also need to record the position of the selector cube, but this time we'll use its position on the board, not its screen position. For example, if the selector is at the centre of the board, then it is in row 4, column 4. To store this information, we'll need another record structure

```

TYPE BoardPosition
  row AS INTEGER
  col AS INTEGER
ENDTYPE

```

then we can declare a variable for the selector's position:

```

selectorposition AS BoardPosition

```

On each turn the player chooses a marble and a position to which the marble is to

be moved. We'll also need to record these two positions:

```
GLOBAL DIM moves(2) AS BoardPosition
```

The final important variable that we will require is a game-state variable. There are three main stages during game play. These are:

- About to choose a marble
- About to choose a move-to space
- Looking at the help text

We need to be aware of which state the game is in, because this dictates what options are currently available to the player. When the player is about to choose a marble, it would be invalid to select an empty pit; however, choosing a pit containing a marble would be invalid when the player is about to choose the move-to position. Trying to play while viewing the Help is also disallowed.

So our new variable will be assigned only the values 1, 2 or 3:

```
gamestate '1-choose marble; 2-choose move-to; 3-showing help
```

Lastly, we need to record how many marbles are currently on the board; when this reduces to one, the game is over. We will do this using a variable named *marbleremaining*.

All the variables above will be made global so that they can be accessed throughout the program:

```
REM *** Declare data structure ***
TYPE Coords
  x AS INTEGER
  z AS INTEGER
ENDTYPE

TYPE BoardPosition
  row AS INTEGER
  col AS INTEGER
ENDTYPE

REM *** Declare main variables ***
GLOBAL DIM board(7,7)                `Marble positions
GLOBAL DIM boardCoords(7,7) AS Coords `Marble coordinates
GLOBAL selectorposition AS BoardPosition `Position of selector
GLOBAL DIM move(2) AS BoardPosition  `marble and move-to
                                      `positions
GLOBAL gamestate                      `Game state
                                      `1-choose marble;
                                      `2-choose space;
                                      `3-showing help
GLOBAL marblesremaining               `marbles on board
```

The images and 3D objects also need to be assigned ID values and it will make the code clearer if these are given names in **#CONSTANT** statements.

```
REM *****
REM *** Constants ***
REM *****
REM *** Object constants ***
#CONSTANT boardobj 1
#CONSTANT selectorobj 2
#CONSTANT firstmarbleobj 3
```



```
#CONSTANT lastmarbleobj      34
REM *** Image constants ***
#CONSTANT boardimg          1
#CONSTANT marbleimg         2
#CONSTANT selectorimg       3
#CONSTANT helpimage         4
REM *** Sprite constants ***
#CONSTANT helpsprite        1
```

Now it's time to have a look at the logic of the game. As usual, for a larger project, the main section should consist mainly of function calls so that we can get an overview of what's going on in just a few lines. In English, the whole logic could be described as follows:

```
Initialise screen
Set up game
REPEAT
    Get player's move
UNTIL game over
```

In the main, this translates into straight function calls but we'll have to know if the player has chosen to quit the game, so the function which gets the player's move will have to return a flag to indicate if the game should be ended. This means that the main section should be coded as:

```
SetUpScreen()
SetUpGame()
REPEAT
    quit = GetPlayersMove()
UNTIL quit OR marblesremaining = 1
END
```

Activity 38.1

Create a new project (*Solitaire.dbpro*) and copy the code for the constants and data into the source file.

Code the main section as given above and add test stubs for each of the functions called. The test stub for *GetPlayersMove()* should be:

```
FUNCTION GetPlayersMove()
    PRINT "GetPlayersMove() executed"
ENDFUNCTION 1
```

Run the program and make sure it executes.

To stop the program finishing before you get time to see all the messages displayed, add a WAIT KEY statement before the END command in the program's main section.

Make sure all the media files have been copied to the project's folder.

Now we'll begin to build up the project by working our way through each routine needed.

Adding *SetUpScreen()*

This routine sets the screen resolution and backdrop as well as positioning the camera, so its code is quite straight forward:

```

FUNCTION SetUpScreen()
  REM *** Set up screen ***
  SET DISPLAY MODE 1280, 1024, 32
  COLOR BACKDROP RGB(255,255,150)
  BACKDROP ON
  REM *** Position camera ***
  AUTOCAM OFF
  POSITION CAMERA 10,10,-20
  POINT CAMERA 0,0,0
ENDFUNCTION

```

Activity 38.2

Replace the test stub for *SetUpScreen()* with the actual code as given above.

Once a 3D object (or sprite) appears on the screen, or we start using a camera, the screen refresh method employed means that the output from any PRINT statement vanishes almost as soon as it appears. Because of this, we can no longer see the messages confirming that routines have been executed.

You might feel that we no longer need to see these messages now that we're up and running but, unfortunately, you're probably wrong! Remember every program is just waiting its opportunity to catch us out - so don't take any chances!

There are various ways we might overcome the problem of the disappearing messages, but in this case we're going to create a replacement for the PRINT statement that displays a string at any point on the screen for a given number of seconds. The new function, *DisplayMessage()* is coded below:

```

FUNCTION DisplayMessage(x,y,message$,seconds#)
  t = TIMER()
  INK 0,0
  millisecs = seconds# * 1000
  WHILE TIMER() - t < millisecs
    SET CURSOR x,y
    PRINT message$
  ENDWHILE
ENDFUNCTION

```

Activity 38.3

Add *DisplayMessage()* to the end of your program and replace all the PRINT statements in the test stubs with calls to this function.

Adding *SetUpGame()*

This routine is a bit more complex since we need to set up the board, add the marbles, position the selector and set up the Help feature. Since each of these will take up several lines of code, it is best if we create new routines to handle each stage. This means that the *SetUpGame()* function itself mainly consists of calls to other functions, but it also initialises the *gamestate* variable:

```

FUNCTION SetUpGame()
  CreateBoard()
  CreateMarbles()
  CreateSelector()
  SetUpHelp()
  gamestate = 1
ENDFUNCTION

```

Activity 38.4

Add the code for *SetUpGame()* to your program and add test stubs for the routines that it calls.

Adding *CreateBoard()*

This routine is responsible for setting up the board. This means, not only does it create a thin 3D box to represent the board, texturing it with the board image shown in FIG-38.2, but it also fills the *board* and *boardCoords* arrays with relevant data.

While the visual aspects of creating the board require only a few lines, initialising the arrays is much more complex, so again we'll make another routine to do that part of the job (*CreateInternalBoard()*). The code for *CreateBoard()* is:

```
FUNCTION CreateBoard()  
    REM *** Set up board arrays ***  
    CreateInternalBoard()  
    REM *** Create and texture board ***  
    MAKE OBJECT BOX boardobj,27,27,0.5  
    LOAD IMAGE "board.bmp",boardimg  
    TEXTURE OBJECT boardobj,boardimg  
    REM *** Turn and position board ***  
    ROTATE OBJECT boardobj,90,0,0  
    POSITION OBJECT boardobj,-0.2,-1.2,0  
ENDFUNCTION
```

Activity 38.5

Update your program with the code above.

Create a test stub for *CreateInternalBoard()*.

Adding *CreateInternalBoard()*

Although this is a rather long routine, it doesn't do anything too complicated; just fills the *board* and *boardCoords* arrays with the values shown earlier. The code for the routine is:

```
FUNCTION CreateInternalBoard()  
    REM *** Initialise the board array ***  
    #CONSTANT EMPTYFIT      0  
    #CONSTANT NOTALLOWED    -1  
    REM *** Assume every position empty ***  
    FOR row = 1 TO 7  
        FOR col = 1 TO 7  
            board(row,col) = EMPTYFIT  
        NEXT col  
    NEXT row  
    REM *** then mark the invalid positions ***  
    board(1,1) = NOTALLOWED  
    board(1,2) = NOTALLOWED  
    board(2,1) = NOTALLOWED  
    board(2,2) = NOTALLOWED  
    board(1,6) = NOTALLOWED  
    board(1,7) = NOTALLOWED  
    board(2,6) = NOTALLOWED  
    board(2,7) = NOTALLOWED  
    board(6,1) = NOTALLOWED  
    board(6,2) = NOTALLOWED
```

```

board(7,1) = NOTALLOWED
board(7,2) = NOTALLOWED
board(6,6) = NOTALLOWED
board(6,7) = NOTALLOWED
board(7,6) = NOTALLOWED
board(7,7) = NOTALLOWED

REM *** Initialise the boardContents array ***
REM *** with the pit coordinates ***
x = -9
z = 9
FOR row = 1 TO 7
  FOR col = 1 TO 7
    BoardCoords(row,col).x = x
    BoardCoords(row,col).z = z
    x = x + 3
  NEXT col
  z = z - 3
  x = -9
NEXT row
ENDFUNCTION

```

Activity 38.6

Add the *CreateInternalBoard()* function to your program.

Although it won't make any visible difference to your program, execute the code just to make sure there are no syntax errors or other problems.

Adding *CreateMarbles()*

Now we need to create the 3D marble objects. These will be placed just above the pits on the board, except for the central pit which remains empty.

The logic for this routine can be described in structured English as:

```

Load marble texture
Set objno to firstmarbleobj
FOR each row on the board DO
  FOR each column on the board DO
    IF the pit is empty AND NOT the central pit THEN
      Create sphere (2 units diameter)
      Texture sphere
      Position sphere at the coordinates given in boardCoords(row,col)
      Store the sphere's ID value, objno, in board(row,col)
      Increment objno
   ENDIF
  ENDFOR
ENDFOR

```

Activity 38.7

Use the logic given above to produce the code for the *CreateMarbles()* function and test your updated program.

Adding *CreateSelector()*

A blue outline cube is used to select which marble is to be moved and its destination. The *CreateSelector()* function creates the cube placing it initially in the central pit. The logic for the routine is:

Load selector texture
 Create selector cube object (1.5 units)
 Texture cube
 Make black areas of cube invisible
 Switch off culling so back lines are drawn
 Position the cube at central pit
 Set *selectorposition* to row 4, column 4

Activity 38.8

Use the logic given to create the final version of the *CreateSelector()* routine and add it to your program.

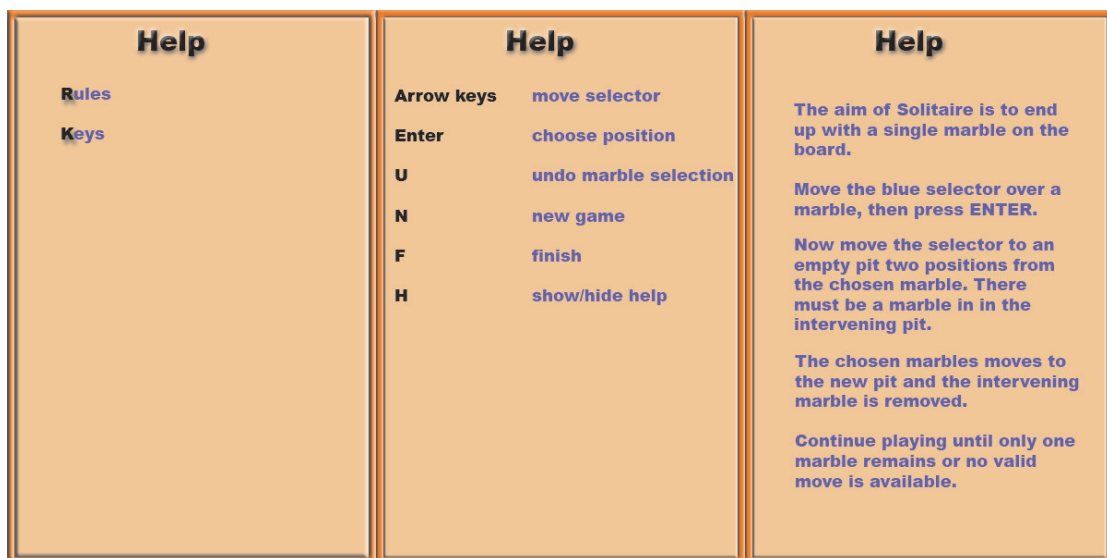
Check that the selector appears at the correct position on the board.

Adding *SetUpHelp()*

The player will be able to get help on the game rules and on which keys can be used by pressing the F1 key (the traditional help key for Microsoft Windows programs).

The Help itself will be created as an animated sprite containing three frames; one for each page of the Help (see FIG-38.6).

FIG-38.6 The Help Image



This routine will create the required sprite, but then hide the sprite. The sprite will appear whenever the F1 key is pressed. If the user presses the F1 key a second time, the Help sprite will disappear.

If you want to create your own Help image using a paint package, remember not to use any black since it will automatically become transparent when the sprite is displayed.

The code for this routine is given below:

```
FUNCTION SetUpHelp()  
  REM *** Create the sprite required by help ***  
  CREATE ANIMATED SPRITE helpsprite, "rules.bmp", 3, 1, helping  
  REM *** Position sprite ***
```

```

        SPRITE helpsprite,10,10,helpimg
    REM *** Set sprite to show first page ***
    SET SPRITE FRAME helpsprite,1
    REM *** Hide sprite ***
    HIDE SPRITE helpsprite
ENDFUNCTION

```

Of course, we'll need another routine later to make the Help sprite appear when required.

Activity 38.9

Add the *SetUpHelp()* function to your program. Check that the program still compiles correctly.

Adding *GetPlayerMove()*

The *GetPlayerMove()* is the core routine of the game, so it will probably come as no surprise that it is the most complex. However, it really only involves reading the keyboard, recognising which key has been pressed and carrying out the required action if it is valid. Invalid actions will produce error messages.

Unlike all the other functions, this one returns a value. If the user has chosen the *quit* option, 1 is returned and zero for all other options.

The logic for this routine is:

```

Set quit to zero
Read a key
IF
    arrow key pressed:
        Move selector
    Enter key pressed:
        IF
            gamestate = 1:
                Select a marble
                IF okay THEN
                    Set gamestate to 2
                ENDIF
            gamestate = 2:
                Select a pit
                IF okay THEN
                    Move marble
                    Set gamestate to 1
                ENDIF
        ENDIF
    U pressed:
        IF gamestate = 2 THEN
            Undo marble selection
            Set gamestate to 1
        ENDIF
    N pressed:
        Restart game
    F1 pressed:
        IF gamestate < 10 THEN
            Show first page of help
            Add 10 to gamestate
        ELSE
            Hide help
            Subtract 10 from gamestate
        ENDIF
    K pressed:
        IF gamestate > 10 THEN
            Show keys help page

```

```

    ENDIF
    R pressed:
        IF gamestate > 10 THEN
            Show rules help page
        ENDIF
    F pressed:
        Set quit to 1
    ENDIF
    Wait for all keys to be released
    Return value of quit

```

The code for this routine is:

```

FUNCTION GetPlayersMove()
    REM *** Assume player will not quit this turn ***
    quit = 0
    REM *** Read a key ***
    REPEAT
        code = SCANCODE()
    UNTIL code <> 0
    SELECT code
        CASE 200,208,205,203      `Arrow keys
            MoveSelector(code)
        ENDCASE
        CASE 28                  `Enter key
            REM *** IF at select marble stage, select marble ***
            IF gamestate = 1
                REM *** IF marble selected, gamestate = 2, ***
                IF SelectMarble()
                    gamestate = 2
                ENDIF
            ELSE
                REM *** IF in select empty pit stage,select pit
                IF gamestate = 2
                    REM *** IF pit okay... ***
                    IF SelectPit()
                        REM *** Move marble and return to state 1
                        MoveMarble()
                        gamestate = 1
                    ENDIF
                ENDIF
            ENDIF
        ENDCASE
        CASE 22                  `U - undo selection
            REM *** IF marble selected THEN ***
            IF gamestate = 2
                REM *** Return marble to normal colour ***
                objno = board(move(1).row,move(1).col)
                TEXTURE OBJECT objno,marbleimg
                REM *** return to gamestate 1 ***
                gamestate = 1
            ENDIF
        ENDCASE
        CASE 49                  `N - new game
            SetUpScreen()
            SetUpGame()
        ENDCASE
        CASE 59                  `F1 - Help
            IF gamestate < 10
                SelectHelpPage(1)
                gamestate = gamestate + 10
            ELSE
                SelectHelpPage(0)
                gamestate = gamestate - 10
            ENDIF
        ENDCASE
        CASE 37                  `K - Help page 2
            IF gamestate > 10

```

```

        SelectHelpPage(2)
    ENDIF
ENDCASE
CASE 19                                `R - Help page 3
    IF gamestate > 10
        SelectHelpPage(3)
    ENDIF
ENDCASE
CASE 33                                `F - Quit game
    quit = 1
ENDCASE
ENDSELECT
REM *** Wait till all keys released ***
WHILE SCANCODE() <> 0
ENDWHILE
ENDFUNCTION quit

```

Activity 38.10

Add the *GetPlayerMove()* function to your program.

Add test stubs for any routines called by *GetPlayerMove()* that are yet to be written.

Test the program.

Adding *MoveSelector()*

The *MoveSelector()* function takes a parameter. This is the arrow key code and is used to decide in which direction the selector is to be moved.

The function's logic is:

```

Set row to copy of selectorposition.row
Set col to copy of selectorposition.col
IF
    left arrow AND not at left hand column :
        Decrement col
    right arrow AND not at right-hand column:
        Increment col
    down arrow AND not at bottom row:
        Decrement row
    up arrow AND not at top row:
        Increment row
ENDIF
IF board(row,col) is a valid position THEN
    Set selectorposition.row to copy of row
    Set selectorposition.col to copy of col
    Move selector to new position
ENDIF

```

and this is coded as:

```

FUNCTION MoveSelector(code)
    REM *** Move selector in response to arrow key ***
    row = selectorposition.row
    col = selectorposition.col
    IF code = 203 AND col>1
        DEC col
    ELSE IF code = 205 AND col < 7
        INC col
    ELSE IF code = 200 AND row > 1
        DEC row
    ELSE IF code = 208 AND row < 7

```



```

        INC row
    ENDIF
ENDIF
ENDIF
ENDIF
REM *** IF a valid position, move selector ***
IF board(row,col) <> -1
    selectorposition.row = row
    selectorposition.col = col
    POSITION OBJECT selectorobj,
    BoardCoords(row,col).x,0,BoardCoords(row,col).z
ENDIF
ENDFUNCTION

```

Activity 38.11

Add the new routine to your program and check that the selector moves correctly.

Adding *SelectMarble()*

In the first phase of a move (when *gamestate* = 1), the player has to select a marble. This involves the player moving the selector over the required pit position and then pressing *Enter*. At this point, control jumps to the *SelectMarble()* routine which checks that the selected pit does, in fact, contain a marble (if not, an error message is displayed) and changes the colour of the marble to indicate its selection. The routine returns 1 when a marble has been selected and zero when no marble was selected.

The logic for this routine is

```

IF selector not over marble THEN
    Display an error message
    Return 0 from function
ENDIF
Change colour of selected marble
Record position of marble in move(1)
Return 1 from function

```

and is coded as:

```

FUNCTION SelectMarble()
    REM *** IF selector not over marble THEN ***
    IF board(selectorposition.row,selectorposition.col) < 1
        REM *** Display message; return 0 ***
        DisplayMessage(100,100,"Choose a marble",2)
        EXITFUNCTION 0
    ENDIF
    REM *** Change marble colour ***
    COLOR OBJECT
    board(selectorposition.row,selectorposition.col),
    RGB(255,255,0)
    REM *** record position of marble ***
    move(1).row = selectorposition.row
    move(1).col = selectorposition.col
ENDFUNCTION 1

```

Activity 38.12

Add *SelectMarble()* to your program and check that it functions correctly.

Adding *SelectPit()*

The second part of a move involves choosing an empty pit into which the selected marble is to be moved.

As well as the obvious check that the selected pit must be empty, we also need to check that it is exactly two pits from the selected marble in either a horizontal or vertical direction. But even that's not all the checking required: moving the marble to the selected pit must involve "jumping over" one other marble (the one that is to be removed).

The logic for the routine is:

```
IF selected pit not empty THEN
    Display "Choose empty pit"
    Return 0 from function
ENDIF
Record selector position in move(2)
IF not a valid move THEN
    Display "Invalid move"
    Return 0 from function
ENDIF
Return 1 from function
```

Checking for a valid move in the final IF mentioned above will be quite complex, so we'll use another function, *isValidMove()*, to carry out the task. This function will return 1 when the move is valid and zero when it's invalid.

Activity 38.13

Using the logic given, add the *SelectPit()* function to your program.

Add a test stub for *isValidMove()* which returns the value 1.

Adding *IsValidMove()*

There are three checks needed in this function:

- That the selected marble is being moved 2 places horizontally
- That the selected marble is being moved 2 places vertically
- That the move involves "jumping" another marble

One of the first two tests must be true and the third test must also be true in order for the move to be a valid one.

Activity 38.14

Write the complete version of the *IsValidMove()* function and test it within your program.

Adding *MoveMarble()*

Once a valid move has been entered, the selected marble must be moved to its new position. On the screen, this not only involves moving the selected 3D sphere and changing it back to its original texture, but also involves removing the "jumped"

marble. The equivalent changes must be made to the *board* array to reflect the new situation. Also, since a marble is removed from the board, the *marblesremaining* variable must be decremented. The logic required by the routine is:

```
Move marble in board array
Move marble on screen
Return marble to original texture
Calculate position of jumped marble
Remove jumped marble from screen
Remove jumped marble from board array
Decrement marblesremaining
```

Activity 38.15

Code the *MoveMarble()* function and add it to your program.

Adding *SelectHelpPage()*

The final routine to be created is the *SelectHelpPage()* function. This function takes a parameter (*pageno*) specifying the page number to be displayed. Since the Help pages are stored as frames in an animated sprite, the *pageno* parameter represents the frame to be displayed. If *pageno* is outside the range 0 to 3, then the function exits; if the *pageno* is zero, the Help sprite is removed from the screen. The function requires the following logic:

```
IF invalid value for pageno THEN
    EXIT
ENDIF
IF pageno > 0 THEN
    Set sprite frame to pageno
ELSE
    Hide Help sprite
ENDIF
```

Activity 38.16

Code the *SelectHelpPage()* from the logic given and test your completed program.

Using the Mouse

Introduction

An alternative way of selecting a marble and the pit it is to be moved to would be to use the mouse. As we saw in the last chapter, the mouse can be used in conjunction with the PICK OBJECT statement to select 3D objects on the screen. However, although this would work fine for selecting the marble, picking the destination pit for the marble would prove more difficult, since there is no object within an empty pit that can be selected with a mouse click.

One way to overcome this problem is to place an invisible object at every valid pit position, since the PICK OBJECT statement can be used to select invisible objects. This means that pits which show a marble will actually contain two objects: a marble and an invisible object; pits without a marble will contain only an invisible object.

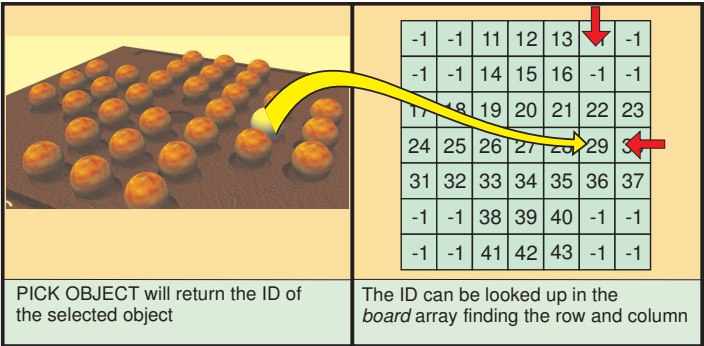
To make a move the player will first click on a pit containing a marble and the mouse position will be compared with the range of marble IDs to discover which one has

been selected. When the destination is being selected as the second part of a move, the mouse position will be compared with the set of invisible objects.

Although this will give us the ID values of the marble and invisible objects, we still have to translate this into row and column values so that the *board* array can be updated and the *move* array can be filled. We can find the selected marble's row and column position by searching the *board* array for a match with the ID returned by the PICK OBJECT statement (see FIG-38.7).

FIG-38.7

Finding the ID of the Selected Marble



If we set up an array containing the IDs of the invisible objects, we can use the same technique to find the row/column position of the empty pit selected as the second part of a move.

Updating the Program

Activity 38.17

Create a new project (*Solitaire2.dbpro*) and copy the source code for the first version of the game into the new source file.

Copy all of the media files used into the new folder.

Since we are using a copy of the original program, it won't matter if things go wrong - we can always go back.

We need to start by removing all references to the selector object since it won't be needed in the new version of the game.

Activity 38.18

Remove all lines containing references to *selectorimg*, *selectorobj*, and *selectorposition* from the program.

In the text we continue to refer to the invisible objects as being hidden.

The next step is to add our invisible objects to each pit. We can't hide the objects since PICK OBJECT doesn't detect hidden objects, but we can texture the objects with a black image and then use SET OBJECT TRANSPARENCY to make the objects disappear.

To do this we'll create some new constants:

```
#CONSTANT hiddenimg      3
#CONSTANT firsthiddenobj 35
#CONSTANT lasthiddenobj  67
```

and an array in which to store the IDs of the invisible objects:

```
GLOBAL DIM PitObjects(7,7)           `IDs of the invisible objects
```

The routine to create the invisible objects is then coded as:

```
FUNCTION CreateHiddenPitObjects()  
  REM *** Set first hidden object ID ***  
  pitobjno = firsthiddenobj  
  REM *** Load image used to texture objects ***  
  LOAD IMAGE "black.bmp",hiddenimg  
  REM *** FOR each position on board ***  
  FOR row = 1 TO 7  
    FOR col = 1 TO 7  
      REM *** IF its a valid position ***  
      IF board(row,col) > -1  
        REM *** Record hidden object ID for that pit  
        PitObjects(row,col) = pitobjno  
        REM *** Make, texture and hide object ***  
        MAKE OBJECT SPHERE pitobjno,2  
        TEXTURE OBJECT pitobjno,hiddenimg  
        POSITION OBJECT pitobjno,boardCoords(row,col).x,  
          0,boardCoords(row,col).z  
        SET OBJECT TRANSPARENCY pitobjno,1  
        REM *** Increment object ID  
        INC pitobjno  
      ENDIF  
    NEXT col  
  NEXT row  
ENDFUNCTION
```

This routine should be called from *SetUpGame()*.

Activity 38.19

Add the constants and global variables mentioned above to your program.

Add the routine *CreateHiddenPitObjects()* and place a call to it in *SetUpGame()*.

Copy the file *black.bmp* to your folder.

Now we have to make some changes to the *GetPlayersMove()* function. The original version of this function reacts to key presses, but in the update we need to allow for both mouse and key presses to be detected. The routine starts by waiting for a key press using the lines

```
REPEAT  
  code = SCANCODE()  
UNTIL code <> 0
```

but in the new version, we need to wait for a key press or a mouse click and this is achieved by the code:

```
REPEAT  
  mouse = MOUSECLICK()  
  code = SCANCODE()  
UNTIL code <> 0 OR mouse <> 0
```

The original SELECT statement that follows was based on the scan code of the key pressed. Since we now have no use for a selector, detecting the arrow keys is no

longer required, so the lines

```
CASE 200,208,205,203    `Arrow keys
    MoveSelector(code)
ENDCASE
```

can be removed from this new version of the program.

The next CASE statement within the SELECT structure is CASE 28 which was used to detect the *Enter* key. But now the mouse button needs to activate this option. To keep our changes to a minimum, we can set *code* to 28 when the mouse is clicked:

```
IF mouse <> 0
    code = 28
ENDIF
```

These lines need to be placed just before the SELECT structure.

At the end of the SELECT structure, we waited for the button pressed by the player to be released using the code

```
REM *** Wait till all keys released ***
WHILE SCANCODE() <> 0
ENDWHILE
```

but now we also need to wait for the mouse button to be released, so our code should be changed to:

```
REM *** Wait till all keys and mouse buttons released ***
WHILE SCANCODE() <> 0 OR MOUSECLICK() <> 0
ENDWHILE
```

Activity 38.20

Make the changes described to the *GetPlayersMove()* routine.

The *SelectMarble()* routine needs to be changed. This version needs to detect the mouse pointer position and select the marble over which it has been placed. The logic for the updated routine is:

```
Find mouse coordinates
Get ID of marble at this screen position
IF no ID detected THEN
    Display "Choose a marble"
    Return 0 from the routine
ENDIF
Find position of ID in board array and record row and column
Change colour of marble selected
Record row and column in moves(1)
Return 1 from routine
```

This is coded as:

```
FUNCTION SelectMarble()
    REM *** Check if position chosen contains marble ***
    x = MOUSEX()
    y = MOUSEY()
    objno = PICK OBJECT(x,y,firstmarbleobj,lastmarbleobj)
    IF objno = 0
        DisplayMessage(100,100,"Choose a marble",2)
        EXITFUNCTION 0
    ENDIF
```

```

    REM *** Find marble's ID in board array ***
    FOR row = 1 TO 7
        FOR col = 1 TO 7
            IF board(row,col) = objno
                prow = row
                pcol = col
            ENDIF
        NEXT col
    NEXT row
    REM *** Change marble colour ***
    COLOR OBJECT objno,RGB(255,255,0)
    REM *** record position of marble
    move(1).row = prow
    move(1).col = pcol
ENDFUNCTION 1 `Valid marble selected

```

The search for the marble ID within the *board* array is very inefficient coding, but for such a small search area will have no real effect on the speed of the game - and it has the advantage of being quite simple.

Activity 38.21

Update your *SelectMarble()* function to match the code given.

The last routine to be changed is the *SelectPit()* function, which again needs to use the mouse's position to determine which empty pit has been selected. The routine's new logic is:

```

Find mouse coordinates
Get ID of hidden object at this screen position
Get ID of marble at this screen position
IF no hidden ID detected OR a marble ID was detected THEN
    Display "Choose an empty space"
    Return 0 from the routine
ENDIF
Find position of hidden object ID in pitObjects array and record the row and column
Record row and column in moves(2)
Check to see if move is valid
IF not valid THEN
    Display "Invalid move"
    Return 0 from routine
ENDIF
Return 1 from routine

```

Activity 38.22

Use the above logic to code the *SelectPit()* function and test your program.

Suggested Enhancements

What we've developed in this chapter is the core of a game, but it still needs all those nice finishing touches that gives a game that professional look.

Some of these would be simple to implement: use a larger font for messages; change the textures used; an introductory splash screen; a finish screen; displaying the number of marbles remaining; displaying the time elapsed; allowing the camera to be moved.

Others would be a bit more complicated: keeping a top 5 best scores (least marbles remaining); allowing the player to backtrack - undoing an unlimited number of

moves (you would need a stack structure for this one).

If you have time, see what enhancements you can come up with - you'll learn a lot from the exercise.

Solutions

Activity 38.1

```
REM *****
REM ***          Constants          ***
REM *****

REM *** Object constants ***
#CONSTANT boardobj      1
#CONSTANT selectorobj   2
#CONSTANT firstmarbleobj 3
#CONSTANT lastmarbleobj 34

REM *** Image constants ***
#CONSTANT boardimg      1
#CONSTANT marbleimg     2
#CONSTANT selectorimg   3
#CONSTANT helping       4

REM *** Sprite constants ***
#CONSTANT helpsprite    1

REM *** Declare data structure ***
TYPE Coords
  x AS INTEGER
  z AS INTEGER
ENDTYPE

TYPE BoardPosition
  row AS INTEGER
  col AS INTEGER
ENDTYPE

REM *** Declare main variables ***
GLOBAL DIM board(7,7) `Marble positions
GLOBAL DIM boardCoords(7,7) AS Coords
`Marble coordinates
GLOBAL selectorposition AS BoardPosition
`Position of selector
GLOBAL DIM move(2) AS BoardPosition
`marble and move-to position
GLOBAL gamestate
`Game state
`1-choose marble;
`2-choose space;
`3-showing help
GLOBAL marblesRemaining
`marbles on board

REM *** Main section ***
SetUpScreen()
SetUpGame()
REPEAT
  quit = GetPlayersMove()
UNTIL quit OR marblesremaining = 1
WAIT KEY `***** Remove later ****
END

`*****
`***** Test stubs*****
`*****
FUNCTION SetUpScreen()
  PRINT "SetUpScreen() executed"
ENDFUNCTION

FUNCTION SetUpGame()
  PRINT "SetUpGame() executed"
ENDFUNCTION

FUNCTION GetPlayersMove()
  PRINT "GetPlayerMove() executed"
ENDFUNCTION 1
```

Activity 38.2

```
REM *****
REM ***          Constants          ***
REM *****

REM *** Object constants ***
#CONSTANT boardobj      1
#CONSTANT selectorobj   2
#CONSTANT firstmarbleobj 3
#CONSTANT lastmarbleobj 34

REM *** Image constants ***
#CONSTANT boardimg      1
#CONSTANT marbleimg     2
#CONSTANT selectorimg   3
#CONSTANT helping       4

REM *** Sprite constants ***
#CONSTANT helpsprite    1

REM *** Declare data structure ***
TYPE Coords
  x AS INTEGER
  z AS INTEGER
ENDTYPE

TYPE BoardPosition
  row AS INTEGER
  col AS INTEGER
ENDTYPE

REM *** Declare main variables ***
GLOBAL DIM board(7,7) `Marble positions
GLOBAL DIM boardCoords(7,7) AS Coords
`Marble coordinates
GLOBAL selectorposition AS BoardPosition
`Position of selector
GLOBAL DIM move(2) AS BoardPosition
`marble and move-to position
GLOBAL gamestate
`Game state
`1-choose marble;
`2-choose space;
`3-showing help
GLOBAL marblesRemaining
`marbles on board

REM *** Main section ***
SetUpScreen()
SetUpGame()
REPEAT
  quit = GetPlayersMove()
UNTIL quit OR marblesremaining = 1
WAIT KEY `***** Remove later ****
END

FUNCTION SetUpScreen()
  REM *** Set up screen ***
  SET DISPLAY MODE 1280, 1024, 32
  COLOR BACKDROP RGB(255,255,150)
  BACKDROP ON
  REM *** Position camera ***
  AUTOCAM OFF
  POSITION CAMERA 10,10,-20
  POINT CAMERA 0,0,0
ENDFUNCTION

`*****
`***** Test stubs*****
`*****
FUNCTION SetUpGame()
  PRINT "SetUpGame() executed"
ENDFUNCTION
```

```

FUNCTION GetPlayersMove()
    PRINT "GetPlayerMove() executed"
ENDFUNCTION 1

```

Activity 38.3

```

REM *****
REM ***          Constants          ***
REM *****

REM *** Object constants ***
#CONSTANT boardobj      1
#CONSTANT selectorobj   2
#CONSTANT firstmarbleobj 3
#CONSTANT lastmarbleobj 34

REM *** Image constants ***
#CONSTANT boardimg      1
#CONSTANT marbleimg     2
#CONSTANT selectorimg   3
#CONSTANT helping       4

REM *** Sprite constants ***
#CONSTANT helpsprite    1

REM *** Declare data structure ***
TYPE Coords
    x AS INTEGER
    z AS INTEGER
ENDTYPE

TYPE BoardPosition
    row AS INTEGER
    col AS INTEGER
ENDTYPE

REM *** Declare main variables ***
GLOBAL DIM board(7,7) `Marble positions
GLOBAL DIM boardCoords(7,7) AS Coords
`Marble coordinates
GLOBAL selectorposition AS BoardPosition
`Position of selector
GLOBAL DIM move(2) AS BoardPosition
`marble and move-to position
GLOBAL gamestate
`Game state
`1-choose marble;
`2-choose space;
`3-showing help
GLOBAL marblesRemaining
`marbles on board

REM *** Main section ***
SetUpScreen()
SetUpGame()
REPEAT
    quit = GetPlayersMove()
UNTIL quit OR marblesremaining = 1
REM *** End program ***
END

FUNCTION SetUpScreen()
    REM *** Set up screen ***
    SET DISPLAY MODE 1280, 1024, 32
    COLOR BACKDROP RGB(255,255,150)
    BACKDROP ON
    REM *** Position camera ***
    AUTOCAM OFF
    POSITION CAMERA 10,10,-20
    POINT CAMERA 0,0,0
ENDFUNCTION

FUNCTION DisplayMessage
(x,y,message$,seconds#)
    t = TIMER()
    INK 0,0
    millisecs = seconds# * 1000

```

```

    WHILE TIMER() - t < millisecs
        SET CURSOR x,y
        PRINT message$
    ENDWHILE
ENDFUNCTION
*****
***** Test stubs*****
*****
FUNCTION SetUpGame()
    DisplayMessage
    (100,100,"SetUpGame() executed",2)
ENDFUNCTION

FUNCTION GetPlayersMove()
    DisplayMessage
    (100,100,"GetPlayersMove() executed",2)
ENDFUNCTION 1

```

The final WAIT KEY statement may also be removed at this stage.

Activity 38.4

```

REM *****
REM ***          Constants          ***
REM *****

REM *** Object constants ***
#CONSTANT boardobj      1
#CONSTANT selectorobj   2
#CONSTANT firstmarbleobj 3
#CONSTANT lastmarbleobj 34

REM *** Image constants ***
#CONSTANT boardimg      1
#CONSTANT marbleimg     2
#CONSTANT selectorimg   3
#CONSTANT helping       4

REM *** Sprite constants ***
#CONSTANT helpsprite    1

REM *** Declare data structure ***
TYPE Coords
    x AS INTEGER
    z AS INTEGER
ENDTYPE

TYPE BoardPosition
    row AS INTEGER
    col AS INTEGER
ENDTYPE

REM *** Declare main variables ***
GLOBAL DIM board(7,7) `Marble positions
GLOBAL DIM boardCoords(7,7) AS Coords
`Marble coordinates
GLOBAL selectorposition AS BoardPosition
`Position of selector
GLOBAL DIM move(2) AS BoardPosition
`marble and move-to position
GLOBAL gamestate
`Game state
`1-choose marble;
`2-choose space;
`3-showing help
GLOBAL marblesRemaining
`marbles on board

REM *** Main section ***
SetUpScreen()
SetUpGame()
REPEAT
    quit = GetPlayersMove()
UNTIL quit OR marblesremaining = 1
REM *** End program ***
END

```

```

FUNCTION SetUpScreen()
  REM *** Set up screen ***
  SET DISPLAY MODE 1280, 1024, 32
  COLOR BACKDROP RGB(255,255,150)
  BACKDROP ON
  REM *** Position camera ***
  AUTOCAM OFF
  POSITION CAMERA 10,10,-20
  POINT CAMERA 0,0,0
ENDFUNCTION

FUNCTION SetUpGame()
  CreateBoard()
  CreateMarbles()
  CreateSelector()
  SetUpHelp()
  gamestate = 1
ENDFUNCTION

FUNCTION DisplayMessage
  ⚡ (x,y,message$,seconds#)
  t = TIMER()
  INK 0,0
  millisecs = seconds# * 1000
  WHILE TIMER() - t < millisecs
    SET CURSOR x,y
    PRINT message$
  ENDWHILE
ENDFUNCTION

`*****
`***** Test stubs*****
`*****

FUNCTION GetPlayersMove()
  DisplayMessage
  ⚡ (100,100,"GetPlayerMove() executed",2)
ENDFUNCTION 1

FUNCTION CreateBoard()
  DisplayMessage(100,100,
  ⚡ "CreateBoard() executed",2)
ENDFUNCTION

FUNCTION CreateMarbles()
  DisplayMessage(100,100,
  ⚡ "CreateMarbles() executed",2)
ENDFUNCTION

FUNCTION CreateSelector()
  DisplayMessage(100,100,
  ⚡ "CreateSelector() executed",2)
ENDFUNCTION

FUNCTION SetUpHelp()
  DisplayMessage(100,100,
  ⚡ "SetUpHelp() executed",2)
ENDFUNCTION

REM *** Sprite constants ***
#CONSTANT helpsprite 1

REM *** Declare data structure ***
TYPE Coords
  x AS INTEGER
  z AS INTEGER
ENDTYPE

TYPE BoardPosition
  row AS INTEGER
  col AS INTEGER
ENDTYPE

REM *** Declare main variables ***
GLOBAL DIM board(7,7) `Marble positions
GLOBAL DIM boardCoords(7,7) AS Coords
`Marble coordinates
GLOBAL selectorposition AS BoardPosition
`Position of selector
GLOBAL DIM move(2) AS BoardPosition
`marble and move-to position
GLOBAL gamestate
`Game state
`1-choose marble;
`2-choose space;
`3-showing help
GLOBAL marblesRemaining
`marbles on board

REM *** Main section ***
SetUpScreen()
SetUpGame()
REPEAT
  quit = GetPlayersMove()
UNTIL quit OR marblesremaining = 1
REM *** End program ***
END

FUNCTION SetUpScreen()
  REM *** Set up screen ***
  SET DISPLAY MODE 1280, 1024, 32
  COLOR BACKDROP RGB(255,255,150)
  BACKDROP ON
  REM *** Position camera ***
  AUTOCAM OFF
  POSITION CAMERA 10,10,-20
  POINT CAMERA 0,0,0
ENDFUNCTION

FUNCTION SetUpGame()
  CreateBoard()
  CreateMarbles()
  CreateSelector()
  SetUpHelp()
  gamestate = 1
ENDFUNCTION

FUNCTION CreateBoard()
  REM *** Set up board arrays ***
  CreateInternalBoard()
  REM *** Create and texture board ***
  MAKE OBJECT BOX boardobj,27,27,0.5
  LOAD IMAGE "board.bmp",boardimg
  TEXTURE OBJECT boardobj,boardimg
  REM *** Turn and position board ***
  ROTATE OBJECT boardobj,90,0,0
  POSITION OBJECT boardobj,-0.2,-1.2,0
ENDFUNCTION

FUNCTION DisplayMessage
  ⚡ (x,y,message$,seconds#)
  t = TIMER()
  INK 0,0
  millisecs = seconds# * 1000
  WHILE TIMER() - t < millisecs
    SET CURSOR x,y

```

Activity 38.5

```

REM *****
REM *** Constants ***
REM *****

REM *** Object constants ***
#CONSTANT boardobj 1
#CONSTANT selectorobj 2
#CONSTANT firstmarbleobj 3
#CONSTANT lastmarbleobj 34

REM *** Image constants ***
#CONSTANT boardimg 1
#CONSTANT marbleimg 2
#CONSTANT selectorimg 3
#CONSTANT helping 4

```

```

PRINT message$
ENDWHILE
ENDFUNCTION

`*****
`***** Test stubs*****
`*****

FUNCTION GetPlayersMove()
  DisplayMessage
  ⤵ (100,100,"GetPlayerMove() executed",2)
ENDFUNCTION 1

FUNCTION CreateMarbles()
  DisplayMessage(100,100,
  ⤵ "CreateMarbles() executed",2)
ENDFUNCTION

FUNCTION CreateSelector()
  DisplayMessage(100,100,
  ⤵ "CreateSelector() executed",2)
ENDFUNCTION

FUNCTION SetUpHelp()
  DisplayMessage(100,100,
  ⤵ "SetUpHelp() executed",2)
ENDFUNCTION

FUNCTION CreateInternalBoard()
  DisplayMessage(100,100,
  ⤵ "CreateInternalBoard() executed",2)
ENDFUNCTION

```

```

GLOBAL marblesRemaining
`marbles on board

REM *** Main section ***
SetUpScreen()
SetUpGame()
REPEAT
  quit = GetPlayersMove()
UNTIL quit OR marblesRemaining = 1
REM *** End program ***
END

FUNCTION SetUpScreen()
  REM *** Set up screen ***
  SET DISPLAY MODE 1280, 1024, 32
  COLOR BACKDROP RGB(255,255,150)
  BACKDROP ON
  REM *** Position camera ***
  AUTOCAM OFF
  POSITION CAMERA 10,10,-20
  POINT CAMERA 0,0,0
ENDFUNCTION

FUNCTION SetUpGame()
  CreateBoard()
  CreateMarbles()
  CreateSelector()
  SetUpHelp()
  gamestate = 1
ENDFUNCTION

FUNCTION CreateBoard()
  REM *** Set up board arrays ***
  CreateInternalBoard()
  REM *** Create and texture board ***
  MAKE OBJECT BOX boardobj,27,27,0.5
  LOAD IMAGE "board.bmp",boardimg
  TEXTURE OBJECT boardobj,boardimg
  REM *** Turn and position board ***
  ROTATE OBJECT boardobj,90,0,0
  POSITION OBJECT boardobj,-0.2,-1.2,0
ENDFUNCTION

FUNCTION CreateInternalBoard()
  REM *** Initialise the board array ***
  #CONSTANT EMPTYPI 0
  #CONSTANT NOTALLOWED -1
  REM *** Assume every position empty ***
  FOR row = 1 TO 7
    FOR col = 1 TO 7
      board(row,col) = EMPTYPI
    NEXT col
  NEXT row
  REM *** then mark invalid positions ***
  board(1,1) = NOTALLOWED
  board(1,2) = NOTALLOWED
  board(2,1) = NOTALLOWED
  board(2,2) = NOTALLOWED
  board(1,6) = NOTALLOWED
  board(1,7) = NOTALLOWED
  board(2,6) = NOTALLOWED
  board(2,7) = NOTALLOWED
  board(6,1) = NOTALLOWED
  board(6,2) = NOTALLOWED
  board(7,1) = NOTALLOWED
  board(7,2) = NOTALLOWED
  board(6,6) = NOTALLOWED
  board(6,7) = NOTALLOWED
  board(7,6) = NOTALLOWED
  board(7,7) = NOTALLOWED
  REM *** Initialise boardContents ***
  REM *** with the pit coordinates ***
  x = -9
  z = 9
  FOR row = 1 TO 7
    FOR col = 1 TO 7
      BoardCoords(row,col).x = x
      BoardCoords(row,col).z = z
    NEXT col
  NEXT row

```

Activity 38.6

```

REM *****
REM *** Constants ***
REM *****

REM *** Object constants ***
#CONSTANT boardobj 1
#CONSTANT selectorobj 2
#CONSTANT firstmarbleobj 3
#CONSTANT lastmarbleobj 34

REM *** Image constants ***
#CONSTANT boardimg 1
#CONSTANT marbleimg 2
#CONSTANT selectorimg 3
#CONSTANT helping 4
REM *** Sprite constants ***
#CONSTANT helpsprite 1

REM *** Declare data structure ***
TYPE Coords
  x AS INTEGER
  z AS INTEGER
ENDTYPE

TYPE BoardPosition
  row AS INTEGER
  col AS INTEGER
ENDTYPE

REM *** Declare main variables ***
GLOBAL DIM board(7,7) `Marble positions
GLOBAL DIM boardCoords(7,7) AS Coords
`Marble coordinates
GLOBAL selectorposition AS BoardPosition
`Position of selector
GLOBAL DIM move(2) AS BoardPosition
`marble and move-to position
GLOBAL gamestate
`Game state
`1-choose marble;
`2-choose space;
`3-showing help

```

```

        x = x + 3
    NEXT col
    z = z - 3
    x = -9
NEXT row
ENDFUNCTION

FUNCTION DisplayMessage
    ⚡ (x,y,message$,seconds#)
    t = TIMER()
    INK 0,0
    millisecs = seconds# * 1000
    WHILE TIMER() - t < millisecs
        SET CURSOR x,y
        PRINT message$
    ENDWHILE
ENDFUNCTION

` *****
` ***** Test stubs *****
` *****

FUNCTION GetPlayersMove()
    DisplayMessage
    ⚡ (100,100,"GetPlayerMove() executed",2)
ENDFUNCTION 1

FUNCTION CreateMarbles()
    DisplayMessage(100,100,
    ⚡ "CreateMarbles() executed",2)
ENDFUNCTION

FUNCTION CreateSelector()
    DisplayMessage(100,100,
    ⚡ "CreateSelector() executed",2)
ENDFUNCTION

FUNCTION SetUpHelp()
    DisplayMessage(100,100,
    ⚡ "SetUpHelp() executed",2)
ENDFUNCTION

```

Activity 38.7

The code for *CreateMarbles()* is

```

FUNCTION CreateMarbles()
    REM *** Load marble texture ***
    LOAD IMAGE "laval.bmp",marbleimg
    REM *** Create marbles ***
    objno = firstmarbleobj
    FOR row = 1 TO 7
        FOR col = 1 TO 7
            IF board(row,col) = 0 AND
                ⚡ ((row <> 4) OR (col <> 4))
                MAKE OBJECT SPHERE objno,2,20,20
                TEXTURE OBJECT objno,marbleimg
                POSITION OBJECT
                ⚡ objno,BoardCoords(row,col).x,0,
                ⚡ boardCoords(row,col).z
                board(row,col) = objno
                INC objno
            ENDIF
        NEXT col
    NEXT row
ENDFUNCTION

```

Add the code to your program immediately after the *CreateInternalBoard()* function.

If you've entered this separately (rather than modified the test stub), don't forget to remove the test stub version of the routine.

Activity 38.8

The code for *CreateSelector()* is

```

FUNCTION CreateSelector()
    REM *** Load selector texture ***
    LOAD IMAGE "selector.bmp",selectorimg
    REM *** Create selector cube ***
    MAKE OBJECT CUBE selectorobj,1.5
    TEXTURE OBJECT selectorobj,selectorimg
    SET OBJECT TRANSPARENCY selectorobj,1
    SET OBJECT CULL selectorobj,0
    REM *** Position cube ***
    POSITION OBJECT selectorobj,0,0,0
    REM *** Record position of selector ***
    selectorposition.row = 4
    selectorposition.col = 4
ENDFUNCTION

```

The routine should be placed immediately after *CreateMarbles()*.

Activity 38.9

Place *SetUpHelp()* immediately after *CreateSelector()*.

Activity 38.10

Place *GetPlayersMove()* after *SetUpHelp()*.

The following test stubs should also be added:

```

FUNCTION MoveSelector(move)
    DisplayMessage(100,100,
    ⚡ "MoveSelector() executed",2)
ENDFUNCTION

FUNCTION SelectMarble()
    DisplayMessage(100,100,
    ⚡ "SelectMarble() executed",2)
ENDFUNCTION 1

FUNCTION SelectPit()
    DisplayMessage(100,100,
    ⚡ "SelectPit() executed",2)
ENDFUNCTION 1

FUNCTION MoveMarble()
    DisplayMessage(100,100,
    ⚡ "MoveMarble() executed",2)
ENDFUNCTION

FUNCTION SelectHelpPage(page)
    DisplayMessage(100,100,
    ⚡ "SelectHelpPage() executed",2)
ENDFUNCTION

```

Activity 38.11

No solution required.

Activity 38.12

No solution required.

Activity 38.13

The code for *SelectPitt()* is:

```

FUNCTION SelectPit()
  REM *** IF selected pit not empty, exit
  IF board(selectorposition.row,
  selectorposition.col) <> 0
    DisplayMessage(100,100,
    ↳ "Choose an empty space",2)
  EXITFUNCTION 0
ENDIF
move(2).row = selectorposition.row
move(2).col = selectorposition.col
IF NOT isValidMove()
  DisplayMessage(100,100,"Invalid move",2)
  EXITFUNCTION 0
ENDIF
ENDFUNCTION 1

```

The new test stub is:

```

FUNCTION isValidMove()
  DisplayMessage(100,100,
  ↳ "isValidMove() executed",2)
ENDFUNCTION 1

```

Activity 38.14

The code for *isValidMove()* is:

```

FUNCTION IsValidMove()
  REM *** Invalid if start & finish pits
  REM *** not exactly 2 pits apart ***
  REM *** or no marble in-between ***
  test1 = (ABS(move(1).row-move(2).row)=2)
  ↳ AND (move(1).col = move(2).col)
  test2 = (ABS(move(1).col-move(2).col)=2)
  ↳ AND (move(1).row = move(2).row)
  midrow = ABS(move(1).row+move(2).row)/2
  midcol = ABS(move(1).col+move(2).col)/2
  test3 = board(midrow,midcol) > 1
  REM *** Return 1 if tests checks okay ***
  IF (test1 OR test2) AND test3
    result = 1
  ELSE `otherwise return zero
    result = 0
  ENDF
ENDFUNCTION result

```

Activity 38.15

The code for *MoveMarble()* is:

```

FUNCTION MoveMarble()
  REM *** Move marble in board array data ***
  board(moves(2).row,moves(2).col) =
  board(moves(1).row,moves(1).col)
  board(moves(1).row,moves(1).col) = 0
  REM *** Move marble on screen ***
  POSITION OBJECT
  ↳ board(moves(2).row,moves(2).col),
  ↳ boardCoords(moves(2).row,moves(2).col).x,
  ↳ 0,boardCoords(moves(2).row,moves(2).col).z
  TEXTURE OBJECT
  ↳ board(moves(2).row,moves(2).col),marbleimg
  REM *** Calc position of jumped marble ***
  jumpedrow = (moves(1).row+moves(2).row)/2
  jumpedcol = (moves(1).col+moves(2).col)/2
  REM *** Remove jumped marble from screen ***
  jumpedmarble = board(jumpedrow,jumpedcol)
  HIDE OBJECT jumpedmarble
  DELETE OBJECT jumpedmarble
  REM *** Remove marble from board array ***
  board(jumpedrow,jumpedcol)=0
  REM *** Decrement marblesremaining ***
  DEC marblesremaining
ENDFUNCTION

```

Activity 38.16

The completed program should be coded as:

```

REM *****
REM *** Constants ***
REM *****

REM *** Object constants ***
#CONSTANT boardobj 1
#CONSTANT selectorobj 2
#CONSTANT firstmarbleobj 3
#CONSTANT lastmarbleobj 34

REM *** Image constants ***
#CONSTANT boarding 1
#CONSTANT marbleimg 2
#CONSTANT selectorimg 3
#CONSTANT helping 4

REM *** Sprite constants ***
#CONSTANT helpsprite 1

REM *** Declare data structure ***
TYPE Coords
  x AS INTEGER
  z AS INTEGER
ENDTYPE

TYPE BoardPosition
  row AS INTEGER
  col AS INTEGER
ENDTYPE

REM *** Declare main variables ***
GLOBAL DIM board(7,7)
`Marble positions
GLOBAL DIM boardCoords(7,7) AS Coords
`Marble coordinates
GLOBAL selectorposition AS BoardPosition
`Position of selector
GLOBAL DIM move(2) AS BoardPosition
`marble and move-to positions
GLOBAL gamestate `Game state
GLOBAL gamestate `1-choose marble;
`2-choose space;
`3-showing help

GLOBAL marblesRemaining
`marbles on board

REM *** Main section ***
SetUpScreen()
SetUpGame()
REPEAT
  quit = GetPlayersMove()
UNTIL quit OR marblesremaining = 1
REM *** End program ***
END

FUNCTION SetUpScreen()
  REM *** Set up screen ***
  SET DISPLAY MODE 1280,1024,32
  COLOR BACKDROP RGB(255,255,150)
  BACKDROP ON
  REM *** Position camera ***
  AUTOCAM OFF
  POSITION CAMERA 10,10,-20
  POINT CAMERA 0,0,0
ENDFUNCTION

FUNCTION SetUpGame()
  CreateBoard()
  CreateMarbles()
  CreateSelector()
  SetUpHelp()
  gamestate = 1
ENDFUNCTION

```

```

FUNCTION CreateBoard()
    REM *** Set up board arrays ***
    CreateInternalBoard()
    REM *** Create and texture board ***
    MAKE OBJECT BOX boardobj,27,27,0.5
    LOAD IMAGE "board.bmp",boardimg
    TEXTURE OBJECT boardobj,boardimg
    REM *** Turn and position board ***
    ROTATE OBJECT boardobj,90,0,0
    POSITION OBJECT boardobj,-0.2,-1.2,0
ENDFUNCTION

FUNCTION CreateInternalBoard()
    REM *** Initialise the board array ***
    #CONSTANT EMPTYPT 0
    #CONSTANT NOTALLOWED -1
    REM *** Assume every position empty ***
    FOR row = 1 TO 7
        FOR col = 1 TO 7
            board(row,col) = EMPTYPT
        NEXT col
    NEXT row
    REM *** then mark invalid positions ***
    board(1,1) = NOTALLOWED
    board(1,2) = NOTALLOWED
    board(2,1) = NOTALLOWED
    board(2,2) = NOTALLOWED
    board(1,6) = NOTALLOWED
    board(1,7) = NOTALLOWED
    board(2,6) = NOTALLOWED
    board(2,7) = NOTALLOWED
    board(6,1) = NOTALLOWED
    board(6,2) = NOTALLOWED
    board(7,1) = NOTALLOWED
    board(7,2) = NOTALLOWED
    board(6,6) = NOTALLOWED
    board(6,7) = NOTALLOWED
    board(7,6) = NOTALLOWED
    board(7,7) = NOTALLOWED
    REM *** Initialise boardContents ***
    REM *** with the pit coordinates ***
    x = -9
    z = 9
    FOR row = 1 TO 7
        FOR col = 1 TO 7
            BoardCoords(row,col).x = x
            BoardCoords(row,col).z = z
            x = x + 3
        NEXT col
        z = z - 3
        x = -9
    NEXT row
ENDFUNCTION

FUNCTION CreateMarbles()
    REM *** Load marble texture ***
    LOAD IMAGE "lava1.bmp",marbleimg
    REM *** Create marbles ***
    objno = firstmarbleobj
    FOR row = 1 TO 7
        FOR col = 1 TO 7
            IF board(row,col) = 0 AND
                ((row <> 4) OR (col <> 4))
                MAKE OBJECT SPHERE objno,2,20,20
                TEXTURE OBJECT objno,marbleimg
                POSITION OBJECT
                objno,BoardCoords(row,col).x,0,
                objno,BoardCoords(row,col).z
                board(row,col) = objno
                INC objno
            ENDIF
        NEXT col
    NEXT row
ENDFUNCTION

```

```

FUNCTION CreateSelector()
    REM *** Load selector texture ***
    LOAD IMAGE "selector.bmp",selectorimg
    REM *** Create selector cube ***
    MAKE OBJECT CUBE selectorobj,1.5
    TEXTURE OBJECT selectorobj, selectorimg
    SET OBJECT TRANSPARENCY selectorobj,1
    SET OBJECT CULL selectorobj,0
    REM *** Position cube ***
    POSITION OBJECT selectorobj,0,0,0
    REM *** Record position of selector ***
    selectorposition.row = 4
    selectorposition.col = 4
ENDFUNCTION

FUNCTION SetUpHelp()
    REM *** Create sprite requ'd by help ***
    CREATE ANIMATED SPRITE helpsprite,
    objno,"rules.bmp",3,1,helpimg
    REM *** Position sprite ***
    SPRITE helpsprite,10,10,helpimg
    REM *** Set sprite to show first page ***
    SET SPRITE FRAME helpsprite,1
    REM *** Hide sprite ***
    HIDE SPRITE helpsprite
ENDFUNCTION

FUNCTION GetPlayersMove()
    REM *** Read a key ***
    REPEAT
        code = SCANCODE()
    UNTIL code <> 0
    quit = 0
    SELECT code
        CASE 200,208,205,203      `Arrow keys
            MoveSelector(code)
        ENDCASE
        CASE 28                  `Enter key
            REM *** IF state 1, get marble ***
            IF gamestate = 1
                REM *** Select marble ***
                REM *** IF okay, move to state 2 ***
                IF SelectMarble()
                    gamestate = 2
                ENDIF
            REM *** If state 2, select pit ***
            ELSE IF gamestate = 2
                REM *** IF okay THEN ***
                IF SelectPit()
                    REM *** Move marble ***
                    MoveMarble()
                    REM *** Return to state 1 ***
                    gamestate = 1
                ENDIF
            ENDIF
        ENDCASE
    CASE 22                      `U - undo selection
        REM *** IF marble selected THEN ***
        IF gamestate = 2
            REM *** return original texture ***
            objno =
            board(move(1).row,move(1).col)
            TEXTURE OBJECT objno,marbleimg
            REM *** Return to state 1 ***
            gamestate = 1
        ENDIF
    ENDCASE
    CASE 49                      `N - new game
        SetUpScreen()
        SetUpGame()
    ENDCASE
    CASE 59                      `F1 - Help
        REM *** IF help screen not showing ***
        IF gamestate < 10
            REM *** Show help page 1 ***
            SelectHelpPage(1)
            REM *** In Help state ***

```



```

        gamestate = gamestate + 10
    ELSE
        SelectHelpPage(0)
        gamestate = gamestate - 10
    ENDIF
ENDCASE
CASE 37      `K - Help page 2
    IF gamestate > 10
        SelectHelpPage(2)
    ENDIF
ENDCASE
CASE 19      `R - Help page 3
    IF gamestate > 10
        SelectHelpPage(3)
    ENDIF
ENDCASE
CASE 33      `F - Quit game
    quit = 1
ENDCASE
ENDSELECT
REM *** Wait till all keys released ***
WHILE SCANCODE() <> 0
ENDWHILE
ENDFUNCTION quit

FUNCTION MoveSelector(code)
    REM *** Move selector using arrow key ***
    row = selectorposition.row
    col = selectorposition.col
    IF code = 203 AND col>1
        DEC col
    ELSE IF code = 205 AND col < 7
        INC col
    ELSE IF code = 200 AND row>1
        DEC row
    ELSE IF code = 208 AND row<7
        INC row
    ENDIF
ENDIF
ENDIF
ENDIF
REM *** IF a valid, move selector ***
IF board(row,col) <> -1
    selectorposition.row = row
    selectorposition.col = col
    POSITION OBJECT
    ⚡selectorobj,BoardCoords(row,col).x,
    ⚡0,BoardCoords(row,col).z
ENDIF
ENDFUNCTION

FUNCTION SelectMarble()
    REM *** IF selector not on marble THEN
    IF board(selectorposition.row,
    ⚡selectorposition.col) < 1
        REM *** Display message; return 0 ***
        DisplayMessage(100,100,
        ⚡"Choose a marble",2)
        EXITFUNCTION 0
    ENDIF
    REM *** Change marble colour ***
    COLOR OBJECT board(selectorposition.row,
    ⚡selectorposition.col),RGB(255,255,0)
    REM *** record position of marble
    move(1).row = selectorposition.row
    move(1).col = selectorposition.col
ENDFUNCTION 1

FUNCTION SelectPit()
    REM *** IF selected pit not empty, exit
    IF board(selectorposition.row,
    ⚡selectorposition.col) <> 0
        DisplayMessage(100,100,
        ⚡"Choose an empty space",2)
        EXITFUNCTION 0
    ENDIF
    move(2).row = selectorposition.row
    move(2).col = selectorposition.col

```

```

    IF NOT isValidMove()
        DisplayMessage(100,100,
        ⚡"Invalid move",2)
        EXITFUNCTION 0
    ENDIF
ENDFUNCTION 1

FUNCTION IsValidMove()
    REM *** Invalid if start and finish pits
    REM *** not exactly 2 distant ***
    REM *** or no marble between ***
    test1 = (ABS(move(1).row-move(2).row)=2)
    ⚡AND (move(1).col = move(2).col)
    test2 = (ABS(move(1).col-move(2).col)=2)
    ⚡AND (move(1).row = move(2).row)
    midrow = ABS(move(1).row+move(2).row)/2
    midcol = ABS(move(1).col+move(2).col)/2
    test3 = board(midrow,midcol) > 1
    REM *** Return 1 if tests okay ***
    IF (test1 OR test2) AND test3
        result = 1
    ELSE
        `otherwise return zero
        result = 0
    ENDIF
ENDFUNCTION result

FUNCTION MoveMarble()
    REM *** Move marble in board array ***
    board(move(2).row,move(2).col) =
    board(move(1).row,move(1).col)
    board(move(1).row,move(1).col) = 0
    REM *** Move marble on screen ***
    POSITION OBJECT
    ⚡board(move(2).row,move(2).col),
    ⚡boardCoords(move(2).row,move(2).col).x,
    ⚡0,boardCoords(move(2).row,move(2).col).z
    TEXTURE OBJECT
    ⚡board(move(2).row,move(2).col),marbleimg
    REM *** Calc jumped marble position ***
    jumpedrow = (move(1).row+move(2).row)/2
    jumpedcol = (move(1).col+move(2).col)/2
    REM *** Remove marble from screen ***
    jumpedmarble = board(jumpedrow,jumpedcol)
    HIDE OBJECT jumpedmarble
    DELETE OBJECT jumpedmarble
    REM *** Remove marble from board ***
    board(jumpedrow,jumpedcol)=0
    REM *** Decrement marblesremaining ***
    DEC marblesremaining
ENDFUNCTION

FUNCTION SelectHelpPage(pageno)
    REM *** IF invalid page selected, exit
    IF pageno < 0 OR pageno > 3
        EXITFUNCTION
    ENDIF
    REM *** IF no zero page no. display page
    IF pageno > 0
        SET SPRITE FRAME helpsprite,pageno
        SHOW SPRITE helpsprite
    ELSE
        REM *** ELSE, hide help ***
        HIDE SPRITE helpsprite
    ENDIF
ENDFUNCTION

FUNCTION DisplayMessage
    ⚡(x,y,message$,seconds#)
    t = TIMER()
    INK 0,0
    millisecs = seconds# * 1000
    WHILE TIMER()-t < millisecs
        SET CURSOR x,y
        PRINT message$
    ENDWHILE
ENDFUNCTION

```


Activity 38.17

No solution required.

Activity 38.18

The following lines should be removed from the main section of the program::

```
#CONSTANT selectorobj      2

#CONSTANT selectorimg      3

GLOBAL selectorposition AS BoardPosition
`Position of selector
```

In *SetUpGame()* remove the line

```
CreateSelector()
```

The function *CreateSelector()* should be deleted.

In *GetPlayersMove()*, remove the line

```
MoveSelector(code)
```

from the first CASE structure.

The routine *MoveSelector()* should be removed.

In *SelectMarble()* remove the lines:

```
IF board(selectorposition.row,
selectorposition.col) < 1

COLOR OBJECT board(selectorposition.row,
↳selectorposition.col), RGB(255,255,0)

move(1).row = selectorposition.row

move(1).col = selectorposition.col
```

In *SelectPit()* remove the lines:

```
IF board(selectorposition.row,
↳selectorposition.col) <> 0

move(2).row = selectorposition.row

move(2).col = selectorposition.col
```

Activity 38.19

The constants and globals should now be coded as:

```
REM *** Constants ***

REM *** Object Constants ***
#CONSTANT boardobj      1
#CONSTANT firstmarbleobj 3
#CONSTANT lastmarbleobj 34
#CONSTANT firsthiddenobj 35
#CONSTANT lasthiddenobj 67

REM *** Image Constants ***
#CONSTANT boardimg      1
#CONSTANT marbleimg     2
#CONSTANT hiddenimg     3
#CONSTANT helpimg       4

REM *** Sprite Constants ***
#CONSTANT helpsprite    1
```

```
REM *** Declare data structure ***
TYPE Coords
  x AS INTEGER
  z AS INTEGER
ENDTYPE

TYPE BoardPosition
  row AS INTEGER
  col AS INTEGER
ENDTYPE

REM *** Declare main variables ***
GLOBAL DIM board(7,7)
`Marble positions on board
GLOBAL DIM BoardCoords(7,7) AS Coords
`Marble world coordinates
GLOBAL DIM moves(2) AS BoardPosition
`marble and move-to positions
GLOBAL gamestate
`Game state 1-choose marble; 2-choose
space; 3-showing help
GLOBAL marblesremaining
`Marbles left on board
GLOBAL DIM PitObjects(7,7)
`ID of the hidden objects
```

Activity 38.20/22

The complete program code is:

```
REM *****
REM *** Constants ***
REM *****
REM *** Object constants ***
#CONSTANT boardobj      1
#CONSTANT firstmarbleobj 3
#CONSTANT lastmarbleobj 34
#CONSTANT firsthiddenobj 35
#CONSTANT lasthiddenobj 67

REM *** Image constants ***
#CONSTANT boardimg      1
#CONSTANT marbleimg     2
#CONSTANT hiddenimg     3
#CONSTANT helping       4
REM *** Sprite constants ***
#CONSTANT helpsprite    1

REM *** Declare data structure ***
TYPE Coords
  x AS INTEGER
  z AS INTEGER
ENDTYPE

TYPE BoardPosition
  row AS INTEGER
  col AS INTEGER
ENDTYPE

REM *** Declare main variables ***
GLOBAL DIM board(7,7)
`Marble positions
GLOBAL DIM boardCoords(7,7) AS Coords
`Marble coordinates
GLOBAL selectorposition AS BoardPosition
`Position of selector
GLOBAL DIM move(2) AS BoardPosition
`marble and move-to positions
GLOBAL gamestate
GLOBAL gamestate `Game state
`1-choose marble;
`2-choose space;
`3-showing help
```

```

GLOBAL marblesRemaining
`marbles on board
GLOBAL DIM pitObjects(7,7)
`IDs of hidden objects

REM *** Main section ***
SetUpScreen()
SetUpGame()
REPEAT
    quit = GetPlayersMove()
UNTIL quit OR marblesremaining = 1
REM *** End program ***
END

FUNCTION SetUpScreen()
    REM *** Set up screen ***
    SET DISPLAY MODE 1280,1024,32
    COLOR BACKDROP RGB(255,255,150)
    BACKDROP ON
    REM *** Position camera ***
    AUTOCAM OFF
    POSITION CAMERA 10,10,-20
    POINT CAMERA 0,0,0
ENDFUNCTION

FUNCTION SetUpGame()
    CreateBoard()
    CreateMarbles()
    CreateHiddenPitObjects()
    SetUpHelp()
    gamestate = 1
ENDFUNCTION

FUNCTION CreateBoard()
    REM *** Set up board arrays ***
    CreateInternalBoard()
    REM *** Create and texture board ***
    MAKE OBJECT BOX boardobj,27,27,0.5
    LOAD IMAGE "board.bmp",boardimg
    TEXTURE OBJECT boardobj,boardimg
    REM *** Turn and position board ***
    ROTATE OBJECT boardobj,90,0,0
    POSITION OBJECT boardobj,-0.2,-1.2,0
ENDFUNCTION

FUNCTION CreateInternalBoard()
    REM *** Initialise the board array ***
    #CONSTANT EMPTYPT 0
    #CONSTANT NOTALLOWED -1
    REM *** Assume every position empty ***
    FOR row = 1 TO 7
        FOR col = 1 TO 7
            board(row,col) = EMPTYPT
        NEXT col
    NEXT row
    REM *** then mark invalid positions ***
    board(1,1) = NOTALLOWED
    board(1,2) = NOTALLOWED
    board(2,1) = NOTALLOWED
    board(2,2) = NOTALLOWED

    board(1,6) = NOTALLOWED
    board(1,7) = NOTALLOWED
    board(2,6) = NOTALLOWED
    board(2,7) = NOTALLOWED
    board(6,1) = NOTALLOWED
    board(6,2) = NOTALLOWED
    board(7,1) = NOTALLOWED
    board(7,2) = NOTALLOWED
    board(6,6) = NOTALLOWED
    board(6,7) = NOTALLOWED
    board(7,6) = NOTALLOWED
    board(7,7) = NOTALLOWED
    REM *** Initialise boardContents ***
    REM *** with the pit coordinates ***
    x = -9
    z = 9

```

```

FOR row = 1 TO 7
    FOR col = 1 TO 7
        BoardCoords(row,col).x = x
        BoardCoords(row,col).z = z
        x = x + 3
    NEXT col
    z = z - 3
    x = -9
NEXT row
ENDFUNCTION

FUNCTION CreateMarbles()
    REM *** Load marble texture ***
    LOAD IMAGE "lava1.bmp",marbleimg
    REM *** Create marbles ***
    objno = firstmarbleobj
    FOR row = 1 TO 7
        FOR col = 1 TO 7
            IF board(row,col) = 0
                AND ((row <> 4) OR (col <> 4))
                MAKE OBJECT SPHERE objno,2,20,20
                TEXTURE OBJECT objno,marbleimg
                POSITION OBJECT objno,
                    boardCoords(row,col).x,0,
                    boardCoords(row,col).z
                board(row,col) = objno
                INC objno
            ENDIF
        NEXT col
    NEXT row
ENDFUNCTION

FUNCTION CreateHiddenPitObjects()
    REM *** Set first hidden object ID ***
    pitobjno = firsthiddenobj
    REM *** Load texture image ***
    LOAD IMAGE "black.bmp",5
    REM *** FOR each position on board ***
    FOR row = 1 TO 7
        FOR col = 1 TO 7
            REM *** IF its a valid position ***
            IF board(row,col)>-1
                REM *** Record hidden object ID
                ↵for that pit
                pitObjects(row,col) = pitobjno
                REM *** Make, texture and hide
                ↵object ***
                MAKE OBJECT SPHERE pitobjno,2
                TEXTURE OBJECT pitobjno,5
                POSITION OBJECT pitobjno,
                    ↵boardCoords(row,col).x,0,
                    ↵boardCoords(row,col).z
                SET OBJECT TRANSPARENCY
                ↵pitobjno,1
                REM *** Increment object ID
                INC pitobjno
            ENDIF
        NEXT col
    NEXT row
ENDFUNCTION

FUNCTION SetUpHelp()
    REM *** Create sprite requ'd by help ***
    CREATE ANIMATED SPRITE helpsprite,
        ↵"rules.bmp",3,1,helpimg
    REM *** Position sprite ***
    SPRITE helpsprite,10,10,helpimg
    REM *** Set sprite to show first page ***
    SET SPRITE FRAME helpsprite,1
    REM *** Hide sprite ***
    HIDE SPRITE helpsprite
ENDFUNCTION

```

```

FUNCTION GetPlayersMove()
    REM *** Read keyboard/mouse ***
    REPEAT
        mouse = MOUSECLICK()
        code = SCANCODE()
    UNTIL code <> 0 OR mouse <> 0
    REM *** Assume not quitting ***
    quit = 0
    REM *** Mouse click same as Enter ***
    IF mouse <> 0
        code = 28
    ENDIF
    SELECT code
        CASE 28                `Enter key
            REM *** IF start of turn THEN ***
            IF gamestate = 1
                REM *** Select marble and
                REM *** IF OK, state 2, ***
                IF SelectMarble()
                    gamestate = 2
                ENDIF
            REM *** IF second stage of turn ***
            ELSE IF gamestate = 2
                REM *** Select pit and
                REM *** IF OK,
                IF SelectPit()
                    REM *** Move marble ***
                    MoveMarble()
                    REM *** Start new turn ***
                    gamestate = 1
                ENDIF
            ENDIF
        ENDIF
    ENDCASE
    CASE 22                `U - undo selection
        IF gamestate = 2
            objno =
            board(move(1).row,move(1).col)
            TEXTURE OBJECT objno,marbleimg
            gamestate = 1
        ENDIF
    ENDCASE
    CASE 49                `N - new game
        SetUpScreen()
        SetUpGame()
    ENDCASE
    CASE 59                `F1 - Help
        IF gamestate < 10
            SelectHelpPage(1)
            gamestate = gamestate + 10
        ELSE
            SelectHelpPage(0)
            gamestate = gamestate - 10
        ENDIF
    ENDCASE
    CASE 37                `K - Help page 2
        IF gamestate > 10
            SelectHelpPage(2)
        ENDIF
    ENDCASE
    CASE 19                `R - Help page 3
        IF gamestate > 10
            SelectHelpPage(3)
        ENDIF
    ENDCASE
    CASE 33                `F - Quit game
        quit = 1
    ENDCASE
ENDSELECT
REM *** Wait keys and mouse released ***
WHILE SCANCODE() <> 0 OR
MOUSECLICK() <> 0
    ENDWHILE
ENDFUNCTION quit

FUNCTION SelectMarble()
    REM *** Check for marble ***
    x = MOUSEX()
    y = MOUSEY()
    objno = PICK OBJECT
    board(x,y,firstmarbleobj,lastmarbleobj)
    REM *** IF no marble ***
    IF objno = 0
        REM *** Show message; return 0
        DisplayMessage(100,100,
            board(x,y,firstmarbleobj,lastmarbleobj),2)
        EXITFUNCTION 0
    ENDIF
    REM *** Find marble's ID in board ***
    FOR row = 1 TO 7
        FOR col = 1 TO 7
            IF board(row,col) = objno
                prow = row
                pcol = col
            ENDIF
        NEXT col
    NEXT row
    REM *** Change marble colour ***
    COLOR OBJECT objno,RGB(255,255,0)
    REM *** record position of marble
    move(1).row = prow
    move(1).col = pcol
ENDFUNCTION 1 `Valid marble selected

FUNCTION SelectPit()
    REM *** Check for hidden obj or marble
    x = MOUSEX()
    y = MOUSEY()
    objno = PICK OBJECT
    board(x,y,firsthiddenobj,lasthiddenobj)
    marbleno = PICK OBJECT(
    board(x,y,firstmarbleobj,lastmarbleobj)
    REM *** IF no hidden obj OR marble THEN
    IF objno = 0 OR marbleno <> 0
        REM *** Display message, return 0 ***
        DisplayMessage(100,100,
            board(x,y,firsthiddenobj,lasthiddenobj),2)
        EXITFUNCTION 0
    ENDIF
    REM *** Find pit's ID in pitObjects ***
    FOR row = 1 TO 7
        FOR col = 1 TO 7
            IF pitObjects(row,col) = objno
                prow = row
                pcol = col
            ENDIF
        NEXT col
    NEXT row
    REM *** Record move-to position ***
    move(2).row = prow
    move(2).col = pcol
    REM *** IF not a valid move THEN ***
    IF NOT isValidMove()
        REM *** Display message;return 0 ***
        DisplayMessage(100,100,
            board(x,y,firsthiddenobj,lasthiddenobj),2)
        EXITFUNCTION 0
    ENDIF
ENDFUNCTION 1 `Valid move-to pit selected

FUNCTION IsValidMove()
    REM *** Invalid if start and finish pits
    REM *** not exactly 2 apart distant ***
    REM *** or no marble in-between ***
    test1 = (ABS(move(1).row-move(2).row)=2)
    AND (move(1).col = move(2).col)
    test2 = (ABS(move(1).col-move(2).col)=2)
    AND (move(1).row = move(2).row)
    midrow = ABS(move(1).row+move(2).row)/2
    midcol = ABS(move(1).col+move(2).col)/2
    test3 = board(midrow,midcol) > 1
    REM *** Return 1 if tests check okay ***
    IF (test1 OR test2) AND test3
        result = 1
    ELSE
        `otherwise return zero
    ENDIF
ENDFUNCTION

```

```

        result = 0
    ENDIF
ENDFUNCTION result

FUNCTION MoveMarble()
    REM *** Move marble in board ***
    board(move(2).row,move(2).col) =
    ⚡board(move(1).row,move(1).col)
    board(move(1).row,move(1).col) = 0
    REM *** Move marble on screen ***
    POSITION OBJECT
    ⚡board(move(2).row,move(2).col),
    ⚡boardCoords(move(2).row,move(2).col).x,0,
    ⚡boardCoords(move(2).row,move(2).col).z
    TEXTURE OBJECT board(move(2).row,move(2).col),
    ⚡marbleimg
    REM *** Calculate position of jumped marble ***
    jumpedrow = (move(1).row+move(2).row)/2
    jumpedcol = (move(1).col+move(2).col)/2
    REM *** Remove jumped marble from screen ***
    jumpedmarble = board(jumpedrow,jumpedcol)
    HIDE OBJECT jumpedmarble
    DELETE OBJECT jumpedmarble
    REM *** Remove jumped marble from board ***
    board(jumpedrow,jumpedcol)=0
    REM *** Decrement marblesremaining ***
    DEC marblesremaining
ENDFUNCTION

FUNCTION SelectHelpPage(pageno)
    IF pageno < 0 OR pageno > 3
        EXITFUNCTION
    ENDIF
    IF pageno > 0
        SET SPRITE FRAME helpsprite,pageno
        SHOW SPRITE helpsprite
    ELSE
        HIDE SPRITE helpsprite
    ENDIF
ENDFUNCTION

FUNCTION DisplayMessage(x,y,message$,seconds#)
    t = TIMER()
    INK 0,0
    millisecs = seconds# * 1000
    WHILE TIMER()-t < millisecs
        SET CURSOR x,y
        PRINT message$
    ENDWHILE
ENDFUNCTION

```

Advanced Lighting and Texturing

Ambient Reflection

Bump Mapping

Diffuse Reflection

Emissive Light Effect

Light Mapping

Shadows

Specular Reflection

Advanced Lighting and Texturing

Introduction

We've seen how basic texturing and lighting can create impressive results, but even more spectacular effects can be created with the various additional statements available in DarkBASIC Pro.

Surface Reflection

In the real world, different materials reflect light in different ways. For example, a red woollen coat, a red shiny metal sphere, and a red plastic box each reflect light differently and hence give us a visual clue as to the material from which the object is constructed. By modifying how a 3D object reflects light we can change its look and so give the impression of different materials.

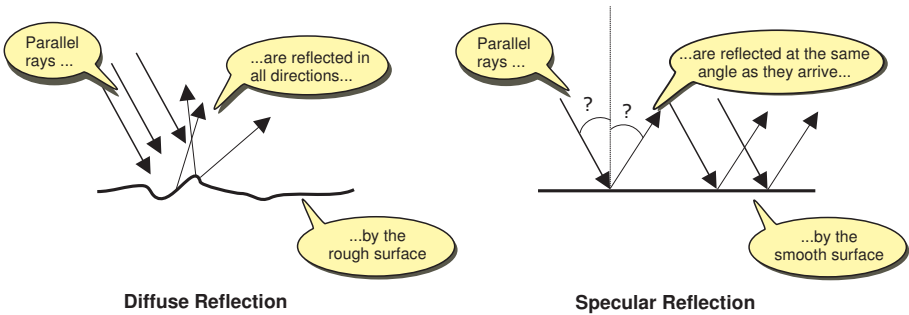
In DarkBASIC Pro there are three types of reflection:

- **ambient reflection** - the reflection of ambient light.
- **diffuse reflection** - the scattered reflection of a directed light. Rough surfaces (even at a microscopic scale) scatter reflected light equally in all directions.
- **specular reflection** - the mirrored reflection of directed light. Light is reflected at the same angle as it arrives at the surface.

FIG-39.1 shows the conceptual ideas behind diffuse and specular reflection.

FIG-39.1

Diffuse and Specular Reflection

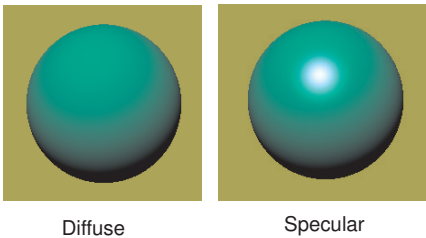


Diffuse reflection creates surfaces that have a matt finish; specular reflection creates a glossy surface.

FIG-39.2 shows the visual effect of both diffuse and specular reflection.

FIG-39.2

Diffuse and Specular Reflection



A surface's reflective properties can be modified by several statements in DarkBASIC Pro.

The SET OBJECT AMBIENT Statement

The simplest of these statements is the SET OBJECT AMBIENT statement which affects how an object reflects ambient light. This statement has the format given in FIG-39.3.

FIG-39.3

The SET OBJECT AMBIENT Statement



In the diagram:

objno

is an integer value giving the ID of the object whose ambient light reflectivity is to be changed.

ambflag

is 0 or 1. If *ambflag* is set to zero, the object reflects no ambient light; if set to 1, the object reflects ambient light normally.

For example, we could switch off object 1's reflection of ambient light using the statement:

```
SET OBJECT AMBIENT 1,0
```

The program in LISTING-39.1 creates a sphere, switches off the directional light, and, after a key press, switches off the sphere's reflection of ambient light.

LISTING-39.1

Manipulating Ambient Light Reflection

```
REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP RGB(20,20,20)
BACKDROP ON

REM *** Programmed screen updates ***
SYNC ON

REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 0,0,-60
POINT CAMERA 0,0,0

REM *** Make cube ***
MAKE OBJECT SPHERE 1,10,50,50

REM *** Switch off directional light ***
HIDE LIGHT 0
SYNC

REM *** Switch off ambient reflection ***
PRINT "Change sphere's reflection of ambient light"
PRINT "Press any key"
SYNC
WAIT KEY
SET OBJECT AMBIENT 1,0
SYNC
SYNC

REM *** End program ***
WAIT KEY
END
```

Activity 39.1

Type in and test the program in LISTING-39.1 (*advanced01.dbpro*).

Modify the program so that the sphere's ambient light reflection is reactivated after another key press.

The SET OBJECT DIFFUSE Statement

How an object produces a reflected diffuse light can be modified using the SET OBJECT DIFFUSE statement. This statement allows the colour of the reflected diffuse light to be specified. The statement has the format shown in FIG-39.4.

FIG-39.4

The SET OBJECT DIFFUSE Statement



In the diagram:

objno

is an integer value giving the ID of the object whose diffuse setting is to be changed.

colour

is an integer value specifying the diffuse colour to be reflected. Where the colour of an object has already been specified, it makes sense to use that same colour value here (this assumes the object is being lit by a white light).

If the light falling on an object is not white, the diffuse colour should be a combination of the object and light colours.

For example, we could create a red diffuse reflection from object 1 using the line:

```
SET OBJECT DIFFUSE 1, RGB (255, 0, 0)
```

Activity 39.2

Create a program showing a blue sphere. Switch off the ambient reflection of the sphere and set its diffuse reflection using cyan.

The SET OBJECT SPECULAR Statement

To recreate the mirror-like reflection of a shiny object, we can use the SET OBJECT SPECULAR statement. The colour of the light reflected from the specular area of an object can also be specified in the statement. The SET OBJECT SPECULAR statement has the format shown in FIG-39.5.

FIG-39.5

The SET OBJECT SPECULAR Statement



In the diagram:

objno

is an integer value giving the ID of the object whose specular reflection setting is to be changed.

colour

is an integer value specifying the specular colour to be used. The colour used should match that of the light shining on the object.

For example, we could create a white specular reflection on object 1 using the line:

```
SET OBJECT SPECULAR 1, RGB(255,255,255)
```

Activity 39.3

Add a yellow spot light - position (-1,3,-5) - to your previous program and create a key-activated yellow specular area on the sphere.

The SET OBJECT SPECULAR POWER Statement

If the SET OBJECT SPECULAR statement has been applied to an object, the amount of specular reflection (shininess) produced can be adjusted using the SET OBJECT SPECULAR POWER statement. This statement has the format shown in FIG-39.6.

FIG-39.6

The SET OBJECT
SPECULAR POWER
Statement



In the diagram:

objno

is an integer value giving the ID of the object whose specular reflection power is to be changed.

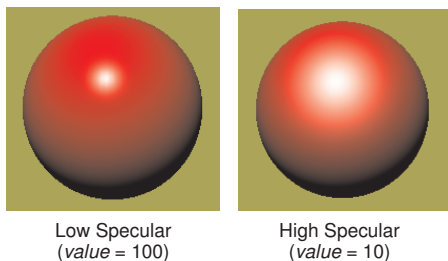
value

is an integer value ranging from 0 to 100. A setting of zero creates specular reflection over a large area of the object; a setting of 100 gives a small reflection area.

The result of using *value* settings of 100 and 10 are shown in FIG-39.7.

FIG-39.7

Varying the Specular
Power Setting



The following code can be used to show the full range of specular reflection:

```
FOR power = 100 TO 0 STEP -1  
  SET OBJECT SPECULAR POWER 1, power  
  WAIT 25  
NEXT power
```

Activity 39.4

Add the code given above to your last program and observe the effect.

The SET OBJECT EMISSIVE Statement

Some objects emit their own light (a bulb, the sun, etc.). Where we want an object to simulate this situation, we can use the SET OBJECT EMISSIVE statement. The

effect of this statement is to colour the object independently from the colour of any light hitting the object. The format for this statement is given in FIG-39.8.

FIG-39.8

The SET OBJECT
EMISSIVE Statement



In the diagram:

objno

is an integer value giving the ID of the object whose emissive light setting is to be changed.

colour

is an integer value specifying the emissive light colour to be used.

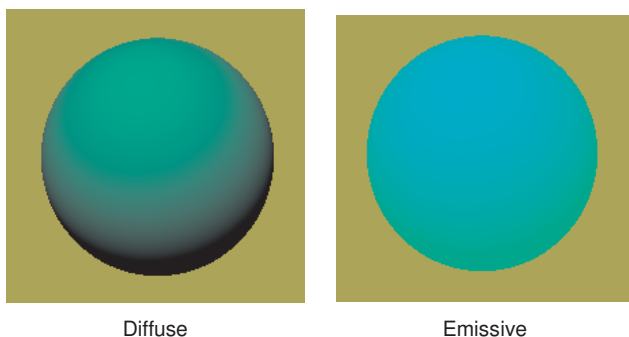
For example, we could create a blue emissive light from object 1 using the line:

```
SET OBJECT EMISSIVE 1,RGB(0,0,255)
```

FIG-39.9 shows the difference between diffuse and emissive lighting.

FIG-39.9

Emissive Lighting Effect



The program in LISTING-39.2 demonstrates this effect by creating a sphere and switching off both the default ambient and directional lights. With no light to reflect, the sphere appears black. After a key press, blue emissive light is assigned to the sphere which then becomes uniformly blue.

LISTING-39.2

Creating Emissive Light

```
REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP RGB(155,155,0)
BACKDROP ON

REM *** position camera ***
AUTOCAM OFF
POSITION CAMERA 0,10,-50

REM *** Create the sphere ***
MAKE OBJECT SPHERE 1,10,100,100

REM *** Switch off lights ***
WAIT KEY
SET AMBIENT LIGHT 0
HIDE LIGHT 0

REM *** Produce fake emissive light ***
WAIT KEY
SET OBJECT EMISSIVE 1, RGB(0,0,255)
REM *** End program ***
WAIT KEY
END
```

Activity 39.5

Type in and test the program in LISTING-39.2 (*advanced02.dbpro*).

It should be noted that the SET OBJECT EMISSIVE statement does not create true light since it will not illuminate other, nearby objects.

Activity 39.6

In your last program, create a 10 unit cube centred on (15,15,20).

Is the cube affected by the blue sphere?

Of course, all we have to do to make the sphere appear to emit light is to place a point light at the centre of the sphere. This light should be the same colour as the sphere's emissive light.

Activity 39.7

In your last program, add the following lines immediately before the final WAIT KEY statement:

```
WAIT KEY
MAKE LIGHT 1
SET POINT LIGHT 1,0,0,0
COLOR LIGHT 1, RGB(0,0,255)
```

Test the program, observing how the new light affects the cube.

When another light shines on an object with emissive properties, the surface of that object is shown in a colour created by a combination of the emissive colour and the external light's colour. So a red directed light shining on a blue emissive sphere will create magenta highlights on the sphere. This is demonstrated in LISTING-39.3.

LISTING-39.3

Combining Emissive and
External Lighting

```
REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP RGB(155,155,0)
BACKDROP ON
REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 0,10,-50

REM *** Switch off ambient light ***
SET AMBIENT LIGHT 0

REM *** Make directional light red ***
COLOR LIGHT 0, RGB(255,0,0)

REM *** Create sphere ***
MAKE OBJECT SPHERE 1,10,100,100

REM *** Create blue emissive effect ***
WAIT KEY
SET OBJECT EMISSIVE 1, RGB(0,0,255)

REM *** End program ***
WAIT KEY
END
```

Activity 39.8

Type in and test the program given in LISTING-39.3 (*advanced03.dbpro*).

The SET OBJECT LIGHT Statement

How an object reacts to lights other than the ambient light can be modified using the SET OBJECT LIGHT statement which has the format given in FIG-39.10.

FIG-39.10

The SET OBJECT
LIGHT Statement



In the diagram:

objno

is an integer value giving the ID of the object whose light reflectivity is to be changed.

lightflag

is 0 or 1. If *lightflag* is set to zero, the object absorbs no light and all surfaces reflect 100% of the light ; if set to 1, the object reflects light normally.

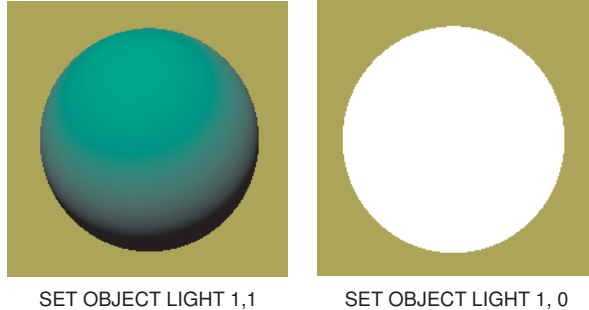
For example, maximum reflection from object 1 would be achieved using the line:

```
SET OBJECT LIGHT 1,0
```

The effects of the two possible settings are shown in FIG-39.11.

FIG-39.11

The Effects of using SET
OBJECT LIGHT



Activity 39.9

Reload program *effects02.dbpro*.

Use the SET OBJECT LIGHT statement to switch off normal light reflection on the cube.

How does this affect the cube?

Change the colour of the ambient light to red. Does this affect the colour reflected by the cube?

Bear in mind that the final reflection created by an object is determined by many factors including the light sources, the object's reflective characteristics, and the texture of the object.

Mappings

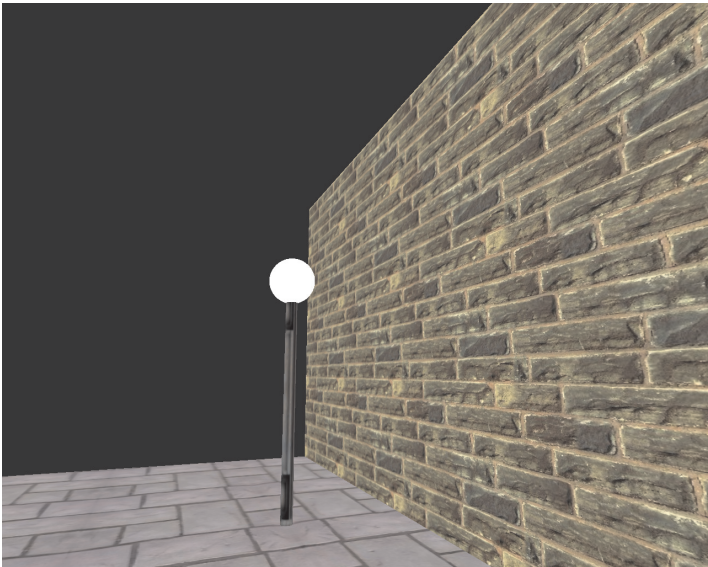
Images can be applied to an object, not to create a basic texture on that object, but to further affect the final appearance of that object.

The SET LIGHT MAPPING ON Statement

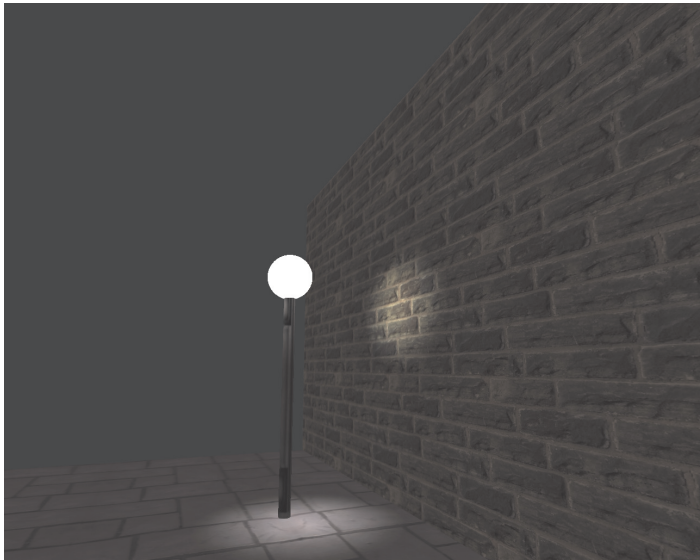
Not all parts of an object's surface reflect equal amounts of light, and although DarkBASIC Pro will take care of this for us automatically, in many cases there are occasions when we need to give it a helping hand. We can modify how parts of an object reflect light using a **light map**. A light map is a grey-scale image which is applied to an object in much the same way as a texture image. Once wrapped onto the object, the light map affects how light is reflected from that object. Areas of the surface which correspond to white parts in the light map image reflect light as normal, while black areas reflect little or no light. The screen dumps in FIG-39.12 show the results of a street at night with and without light mapping.

FIG-39.12

Using Light Mapping



No light mapping applied



Light mapping applied

The default directional light is switched off and ambient light remains on in both scenes. In the second scene, light maps have been applied to the wall, street and lamppost.

The function, *CreateScene()*, which creates the first version of the scene as shown in FIG-39.12 is given below:

```
FUNCTION CreateScene()
    REM *** Create cobbled street ***
    MAKE OBJECT PLAIN 1,30,30
    XROTATE OBJECT 1,90
    LOAD IMAGE "cobblelarge.jpg",1
    TEXTURE OBJECT 1,1
    POSITION OBJECT 1,0,-10,0
    REM *** Create wall ***
    MAKE OBJECT PLAIN 2,30,20
    LOAD IMAGE "brickslarge.jpg",2
    TEXTURE OBJECT 2,2
    POSITION OBJECT 2,0,-1,5
    REM *** Create lamppost ***
    MAKE OBJECT CYLINDER 3,10
    SCALE OBJECT 3, 5,100,5
    POSITION OBJECT 3,0,-5,0
    LOAD IMAGE "lamppost.jpg",3
    TEXTURE OBJECT 3,3
    MAKE OBJECT SPHERE 4,2,100,100
    REM *** Make sphere appear to glow ***
    SET OBJECT EMISSIVE 4,RGB(255,255,255)
    REM *** Put light in sphere ***
    MAKE LIGHT 1
    SET POINT LIGHT 1,0,0,0
ENDFUNCTION
```

Activity 39.10

Write a program to execute the function given above (*lightmap.dbpro*).

The main section should include the following logic:

Set the backdrop colour to RGB(20,20,20) - a dull grey.
Position the camera at (20,-4,-5) pointing at (0,0,3)

Test the program and observe the results.

The light map images used to create the second version of the scene are shown in FIG-39.13.

FIG-39.13

Examples of Light Maps



Light map for wall and street
(*lightmap.jpg*)

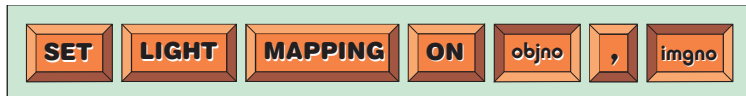


Light map for lamppost
(*lampmap.jpg*)

We can apply light mapping to a specific object using the SET LIGHT MAPPING ON statement which has the format shown in FIG-39.14.

FIG-39.14

The SET LIGHT
MAPPING ON Statement



In the diagram:

objno

is an integer value specifying the ID of the object to which the light map is to be applied.

imgno

is an integer value giving the ID of the image object containing the light map picture.

The program in LISTING-39.4 uses the scene produced by *CreateScene()* and the light maps shown in FIG-39.13 to create the effect shown in the second image of FIG-39.12.

LISTING-39.4

Using a Light Map

```
REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP RGB(20,20,20)
BACKDROP ON

REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 20,-4,-5
POINT CAMERA 0,0,0

REM *** Make scene ***
CreateScene()
REM *** Load and apply light map to street and wall ***
WAIT KEY
LOAD IMAGE "lightmap.jpg",4
SET LIGHT MAPPING ON 1,4
SET LIGHT MAPPING ON 2,4

REM *** Load light map for lamppost ***
WAIT KEY
LOAD IMAGE "lampmap.jpg",5
SET LIGHT MAPPING ON 3,5

REM *** End program ***
WAIT KEY
END

REM ***** Code for CreateScene goes here *****
```

Activity 39.11

Type in and test the program given above (*advanced04.dbpro*). Remember to include the actual code for function *CreateScene()*.

Load and examine the file *lightgrey.jpg*.

Modify your program to use this file as the light map for the wall and ground.

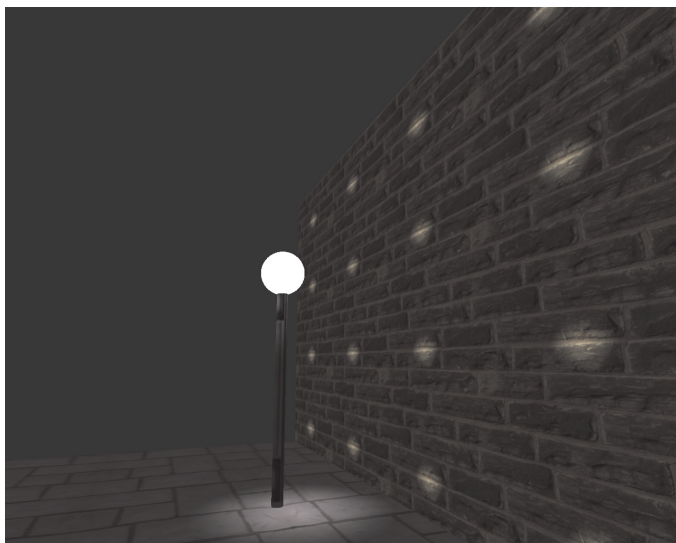
How does this affect the result?

Unfortunately, if a 3D object is textured using tiling, the light map will also be tiled.

The result of a tiled texture on the wall as shown in FIG-39.15 demonstrates the problem.

FIG-39.15

Problems with Tiling and
Light Maps



It's unlikely that we'll want this effect, so we must make sure that any object to which we are going to apply a light map is textured by a single, untiled, image.

The SET BUMP MAPPING ON Statement

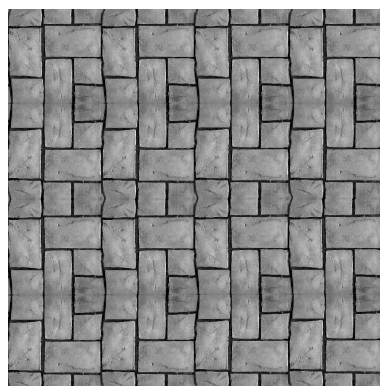
Every 3D object we create is perfectly smooth, whereas many real-life objects are not. Although we can give some impression of roughness by applying a texture to an object, that doesn't actually affect how the object reflects light. A better way to simulate some roughness on a surface is to use a **bump map**. Again, this is a greyscale image which is applied to an object and is used to give the impression of the lumps and bumps that real surfaces contain. Normally, the bump map will be a greyscale version of the original image used to texture an object. For example, the texture and corresponding bump map for the cobbled street in our last program is shown in FIG-39.16.

FIG-39.16

Bump Maps are Often
Based on the Texture
Image



Texture image

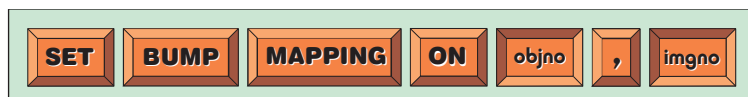


Bump map image

To add a bump map to a 3D object, we use the SET BUMP MAPPING ON statement which has the format shown in FIG-39.17.

FIG-39.17

The SET BUMP
MAPPING ON Statement



In the diagram:

<i>objno</i>	is an integer value specifying the ID of the object to which the bump map is to be applied.
<i>imgno</i>	is an integer value giving the ID of the image object containing the bump map picture.

For example, we could use the file named *cobblesbump.jpg* as a bump map for the cobbled street in our previous program with the lines:

```
LOAD IMAGE "cobblesbump.jpg", 6
SET BUMP MAPPING ON 1, 6
```

Activity 39.12

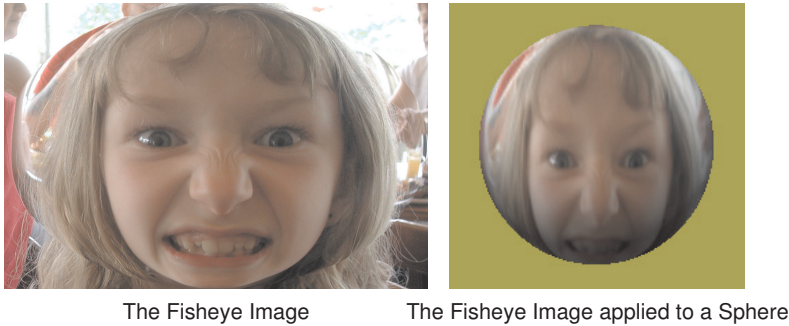
Add the lines given above, preceded by a WAIT KEY statement, to your last program.

How does this affect the appearance of the street?

The SET SPHERE MAPPING ON Statement

A shiny spherical object can mirror the items that are positioned nearby. A quick way to achieve this affect without the processing power that would be needed if it was done for real is to use the SET SPHERE MAPPING ON statement. This allows a fisheye-style image (see FIG-39.18) to be mapped onto a spherical object.

FIG-39.18
A Fisheye Image



The statement has the format shown in FIG-39.19.

FIG-39.19
The SET SPHERE MAPPING ON Statement



In the diagram:

<i>objno</i>	is an integer value specifying the ID of the object to which the sphere map is to be applied.
<i>imgno</i>	is an integer value giving the ID of the image object containing the sphere map picture.

Since a sphere map is meant to represent the reflection of objects on the surface of a sphere, it only makes sense to apply this mapping to a spherical object. At first glance, we may be tempted to think that the overall effect of this statement is really

no different from that of TEXTURE OBJECT, but the reflection shown on the surface of a sphere will not change when the sphere is rotated. A texture, on the other hand, being part of the object, will turn with the sphere. This is demonstrated in LISTING-39.5.

LISTING-39.5

Reflection in a Sphere

```
REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
AUTOCAM OFF
POSITION CAMERA 0,10,-60
POINT CAMERA 0,0,0

REM *** Make sphere ***
MAKE OBJECT SPHERE 1,30,50,50
WAIT KEY

REM *** Load and apply sphere map ***
LOAD IMAGE "fish.jpg",1
SET SPHERE MAPPING ON 1,1
WAIT KEY

REM *** Rotate sphere ***
DO
    TURN OBJECT LEFT 1,1
    WAIT 10
LOOP
REM *** End program ***
END
```

Activity 39.13

Type in and test the program in LISTING-39.5 (*advanced05.dbpro*).

Load the image *stripes.jpg* and use it to texture the sphere. The sphere should now have a texture and sphere map image.

How does this affect the overall result when the sphere is rotated?

However, although the image displayed by sphere mapping does not change when the object rotates, if the object begins to change position, then to maintain the charade of being a reflection, the sphere mapped image needs to change its perspective. This effect is demonstrated by the program in LISTING-39.6.

LISTING-39.6

The Reflection Changes
as the Object Moves

```
REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
AUTOCAM OFF
POSITION CAMERA 0,10,-120
POINT CAMERA -30,0,0
REM *** Make sphere ***
MAKE OBJECT SPHERE 1,30,50,50

REM *** Load and apply sphere map ***
LOAD IMAGE "fish.jpg",1
SET SPHERE MAPPING ON 1,1

REM *** Rotate sphere and move ***
POINT OBJECT 1, 0,1,0
DO
    TURN OBJECT LEFT 1,1
    MOVE OBJECT 1, 1
    WAIT 20
LOOP
REM *** End program ***
END
```

Activity 39.14

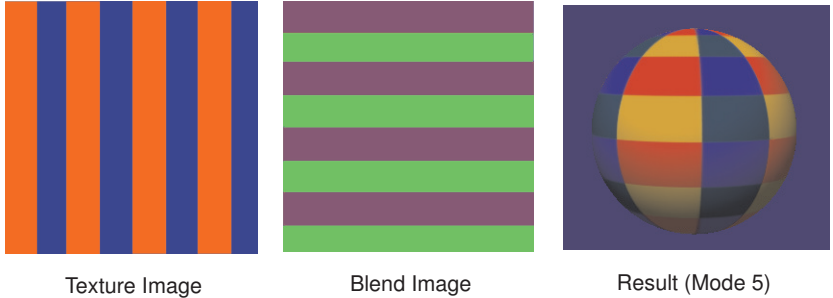
Type in and test the program in LISTING-39.6 (*advanced06.dbpro*).

The SET BLEND MAPPING ON Statement

Blend mapping allows the original texture image and a second image to be merged in various ways to create a texture based on both images. An example of this is shown in FIG-39.20 where the original texture, the blend image and the final result are displayed.

FIG-39.20

One Result Obtained by
Blending Two Images



We can achieve this effect in DarkBASIC Pro by using the SET BLEND MAPPING ON statement which takes the format shown in FIG-39.21.

FIG-39.21

The SET BLEND
MAPPING ON
Statement



In the diagram:

objno

is an integer value specifying the ID of the object to which the blend map is to be applied.

imgno

is an integer value giving the ID of the image object containing the blend map picture.

mode

is an integer value giving the blend option to be used. The actual range for this value is not given but probably lies in the range 1 to 11.

The program in LISTING-39.7 blends the two images shown above, switching mode (FROM 1 TO 11) every 5 seconds.

LISTING-39.7

Blend Mapping

```
REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
AUTOCAM OFF
POSITION CAMERA 0,10,-120
POINT CAMERA 0,0,0

REM *** Make sphere ***
MAKE OBJECT SPHERE 1,30,50,50

REM *** Load texture and blend images ***
LOAD IMAGE "vstripes.jpg",1
LOAD IMAGE "hstripes.jpg",2
```

continued on next page

LISTING-39.7
(continued)

Blend Mapping

```
REM *** Texture sphere ***
TEXTURE OBJECT 1,1
REM *** Try each mode ***
FOR mode = 1 TO 11
  REM *** Change blend mode ***
  SET BLEND MAPPING ON 1,2,mode
  REM *** Rotate sphere for 5 seconds ***
  time = TIMER()
  REM *** Rotate for 5 seconds ***
  WHILE timer() - time < 5000
    TURN OBJECT LEFT 1,1
    REM *** Display current mode ***
    SET CURSOR 100,100
    PRINT "Mode ",mode
  ENDWHILE
NEXT mode
REM *** End program ***
END
```

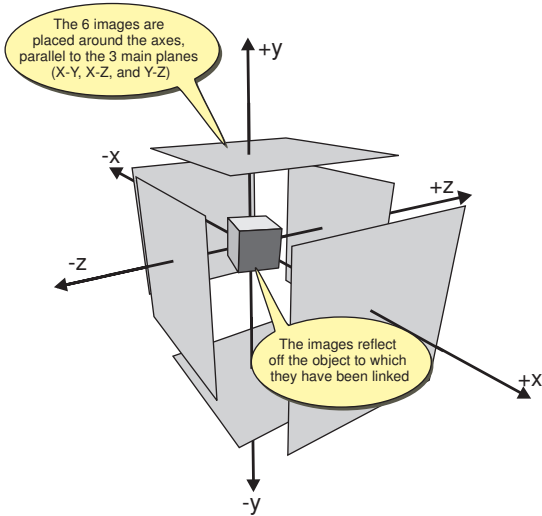
Activity 39.15

Type in and test the program in LISTING-39.7 (*advanced07.jpg*).

The SET CUBE MAPPING ON Statement

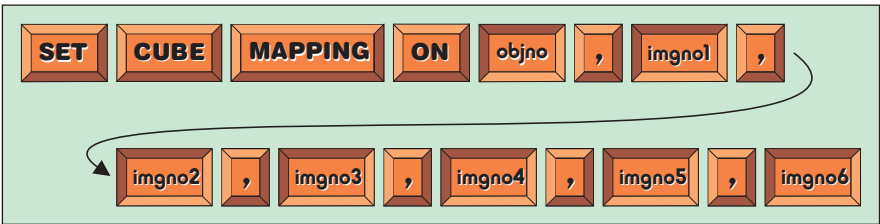
Imagine a cube constructed from mirrors floating in the exact centre of a cubic room. Each wall of the room has a different image painted on it. The floating cube begins to rotate and, as it does so, the reflection from its mirrored sides change. This is the setup we create when using the SET CUBE MAPPING ON statement. Six images, invisible to us, are placed around the specified 3D object and reflected off that object (see FIG-39.22).

FIG-39.22
How Cube Mapping Works



The format for the SET CUBE MAPPING ON statement is given in FIG-39.23.

FIG-39.23
The SET CUBE MAPPING ON Statement



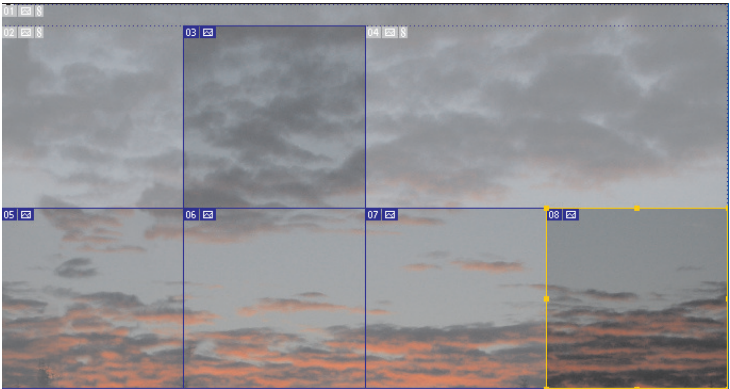
In the diagram:

<i>objno</i>	is an integer value giving the ID of the 3D object to which cube mapping is to be applied.
<i>imgno1, imgno2</i>	are a set of integer values specifying the images to be used on the positive and negative sides of the x-axis respectively.
<i>imgno3, imgno4</i>	are a set of integer values specifying the images to be used on the positive and negative sides of the y-axis respectively.
<i>imgno5, imgno6</i>	are a set of integer values specifying the images to be used on the positive and negative sides of the z-axis respectively.

NOTE: The images used must have dimensions which are some power of 2 (for example, 64, 128, 256, etc.) otherwise the statement will not work.

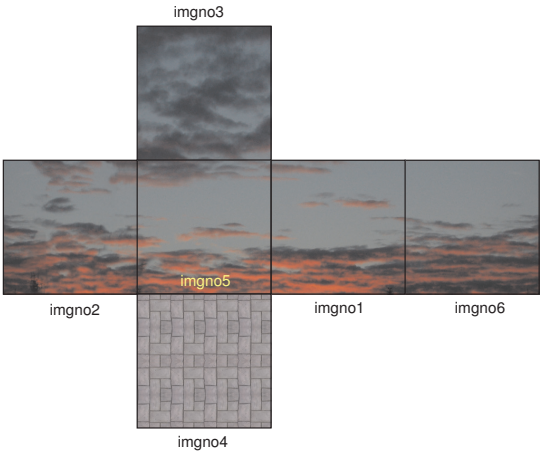
Often the images used will be sky. If a sky sphere is being used, the images used here can be copied from areas of that sky sphere's own image. In FIG-39.24 we see how Adobe ImageReady has been used to slice a sky image into five 512 by 512 pixel images.

FIG-39.24
The Original Sky Image



The sixth image, which represents the base of our 3D space can be the cobbles we used earlier, but modified to a 512 by 512 size. All the images to be used are identified in FIG-39.25.

FIG-39.25
The Sections Used in the
Cube Mapping



The program in LISTING-39.8 demonstrates the use of cube mapping, rotating a cube which shows the reflection of the sky images around it.

LISTING 39.8

Using Cube Mapping

```
REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
AUTOCAM OFF
POSITION CAMERA 0,0,-50
POINT CAMERA 0,0,0

REM *** Add extra light ***
MAKE LIGHT 1
POSITION LIGHT 1,20,-20,50

REM *** Make cube ***
MAKE OBJECT CUBE 22,10

REM *** Load cube mapping images ***
LOAD IMAGE "sky01.jpg",1
LOAD IMAGE "sky02.jpg",2
LOAD IMAGE "sky03.jpg",3
LOAD IMAGE "cobble04.jpg",4
LOAD IMAGE "sky05.jpg",5
LOAD IMAGE "sky06.jpg",6

REM *** Use cube mapping ***
SET CUBE MAPPING ON 22,1,2,3,4,5,6

REM *** Rotate cube using random movements ***
DO
    TURN OBJECT LEFT 22,RND(200)/100.0
    PITCH OBJECT UP 22,RND(200)/100.0
    ROLL OBJECT RIGHT 22,RND(200)/100.0 - 1
    WAIT 10
LOOP

REM *** End program ***
END
```

Activity 39.16

Type in and test the program given in LISTING-39.8 (*advanced08.dbpro*).

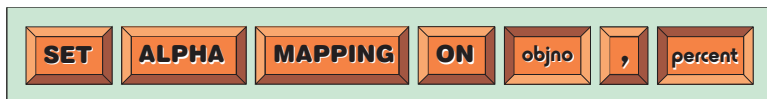
Replace the rotating cube with a cone and see how this affects the result.

The SET ALPHA MAPPING ON Statement

We've already come across one method for making an object appear transparent (SET GHOST ON), but true transparency can be achieved using the SET ALPHA MAPPING ON statement which also allows us to set the degree of transparency displayed by the object. This statement has the format shown in FIG-39.26.

FIG-39.26

The SET ALPHA
MAPPING ON Statement



In the diagram:

objno

is an integer value giving the ID of the 3D object to which alpha mapping is to be applied.

percent

is a real value giving the percentage of opacity required. 0 - invisible; 100 - opaque.

For example, object 1 would be made semi-transparent using the line:

```
SET ALPHA MAPPING ON 1, 50
```

The program in LISTING-39.9 places a rotating cube in front of a textured plane; the cube changes from opaque to invisible as its alpha setting moves from 100 to 0.

LISTING-39.9

Using Alpha Mapping

```
REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
AUTOCAM OFF
POSITION CAMERA 0,0,-100
POINT CAMERA 0,0,0

REM *** Create background object ***
MAKE OBJECT PLAIN 1, 70,100
LOAD IMAGE "brickslarge.jpg",1
TEXTURE OBJECT 1,1

REM *** Create and texture cube ***
MAKE OBJECT CUBE 2,20
LOAD IMAGE "grid8by8b.bmp",2
TEXTURE OBJECT 2,2

REM *** Initialise transparency setting ***
trans = 100

REM *** Rotate object ***
REPEAT
    TURN OBJECT LEFT 2,1
    SET ALPHA MAPPING ON 2,trans
    REM *** 1 in 11 chance of decreasing trans setting ***
    IF RND(10) = 10
        DEC trans
    ENDIF
UNTIL trans = 0

REM *** End program ***
WAIT KEY
END
```

Activity 39.17

Type in and test the program in LISTING-39.9 (*advanced09.dbpro*).

Notice that any part of the textured object which is the transparency colour is invisible irrespective of the opacity setting.

The transparency colour defaults to black but can be changed using the SET IMAGE COLORKEY statement.

Shadows

The SET SHADOW SHADING ON Statement

A 3D object can be made to cast a shadow using the SET SHADOW SHADING ON statement. This statement has the format shown in FIG-39.27.

FIG-39.27 The SET SHADOW SHADING ON Statement



In the diagram:

objno

is an integer value giving the ID of the 3D object which is to cast a shadow.

meshno

is an integer value specifying any mesh used as the shadow. When no mesh is used, this parameter should be set to -1.

distance

is an integer value specifying the maximum distance over which the shadow should be calculated. Objects beyond this distance will not show shadows cast by the object. The *distance* setting is ignored when the graphics hardware is used to perform shadow calculations (see next parameter).

hardflag

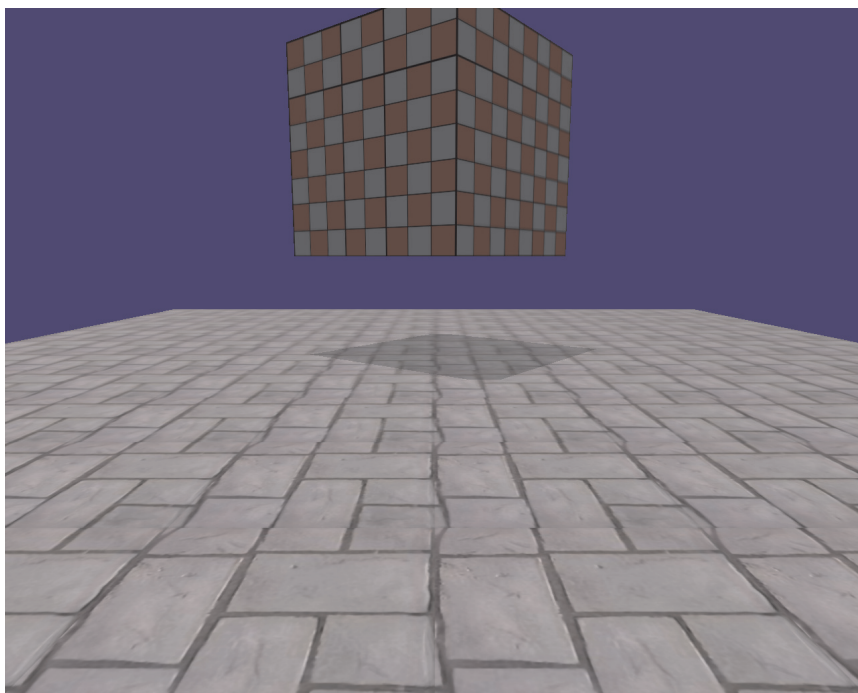
0 or 1. When set to zero, the shadow calculations are performed by the graphics card hardware; a setting of 1 uses the computer's own processor to perform the calculations.

When the last three parameters are omitted, the graphics card hardware is used to calculate shadows. Not all video cards will be able to perform this operation. Creating shadows requires a great deal of processing power, and the frame rate will drop significantly when this option is used.

An example of the shadow effect, a cube casting a shadow on a plane, is shown in FIG-39.28.

FIG-39.28

The Shadow Effect



The program in LISTING-39.10 demonstrates shadowing using the simplest version of the SET SHADOW SHADING ON statement.

LISTING-39.10

Creating Shadows

```
REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
AUTOCAM OFF
POSITION CAMERA 0,10,-60
POINT CAMERA 0,0,0

REM *** Create background object ***
MAKE OBJECT PLAIN 1, 100,100
LOAD IMAGE "cobblestones.jpg",1
TEXTURE OBJECT 1,1
SCALE OBJECT TEXTURE 1,10,10
XROTATE OBJECT 1, -90

REM *** Create and texture cube ***
MAKE OBJECT CUBE 2,20
LOAD IMAGE "grid8by8.bmp",2
TEXTURE OBJECT 2,2
POSITION OBJECT 2,0,20,0
SET POINT LIGHT 0,-10,100,0

REM *** Switch shading on for cube ***
SET SHADOW SHADING ON 2

REM *** Rotate cube and display frame rate ***
DO
    TURN OBJECT LEFT 2,1
    SET CURSOR 100,100
    PRINT "Frame rate : ",SCREEN FPS()
LOOP

REM *** End program ***
END
```

Activity 39.18

Type in and test the program in LISTING-39.10 (*advanced10.dbpro*).

Modify the SET SHADOW SHADING ON statement to include the additional parameters. Set *meshno* to -1, *distance* to 100, and *hardflag* to 1.

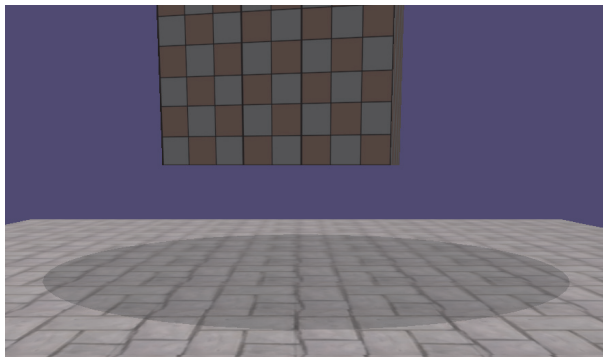
What effect does this change have on the program?

Now give the *distance* parameter a value of 10. How does this affect the shadow?

Normally, the shadow cast will be the shape of the object, but it is possible to cast a shadow of a different shape by creating a mesh and using that mesh's ID as the *meshno* parameter in the SET SHADOW SHADING ON statement (see FIG-39.29).

FIG-39.29

The Cube Casts a Round Shadow!



Activity 39.19

In *advanced10.dbpro*, add the following lines immediately before the SET SHADOW SHADER ON statement :

```
REM *** Create mesh ***  
MAKE OBJECT SPHERE 3,30,50,50  
MAKE MESH FROM OBJECT 1,3  
DELETE OBJECT 3
```

Change the SET SHADOW SHADER ON statement to read:

```
SET SHADOW SHADING ON 2,1,100,1
```

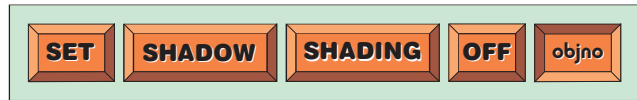
How does this affect the result?

The SET SHADOW SHADING OFF Statement

We can turn off shadow creation for a specific object using the SET SHADOW SHADING OFF statement which has the format shown in FIG-39.30.

FIG-39.30

The SET SHADOW SHADING OFF Statement



In the diagram:

objno

is an integer value giving the ID of the object whose shadow effect is to be switched off.

Activity 39.20

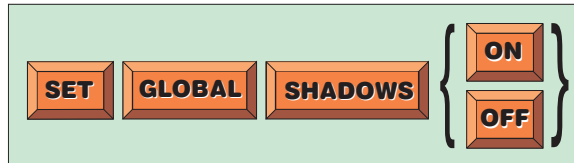
In your last program, switch off the cube's shadow effect when a key is pressed.

The SET GLOBAL SHADOWS Statement

Rather than switch off the shadow of a single object, we can switch off all shadows using the SET GLOBAL SHADOWS statement. The same statement can be used to switch those shadows back on. The SET GLOBAL SHADOWS statement has the format shown in FIG-39.31.

FIG-39.31

The SET GLOBAL SHADOWS Statement



Use SET GLOBAL SHADOW OFF to switch off shadows; use SET GLOBAL SHADOWS ON to re-instate all shadows.

The program in LISTING-39.11 is a variation on LISTING-39.8 but allows the cube's shadow to be switched on by pressing the "s" key and off by pressing "o".

LISTING-39.11

Switching Shadows On
and Off

```
REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
AUTOCAM OFF
POSITION CAMERA 0,10,-60
POINT CAMERA 0,0,0

REM *** Create background object ***
MAKE OBJECT PLAIN 1, 100,100
LOAD IMAGE "cobblestones.jpg",1
TEXTURE OBJECT 1,1
SCALE OBJECT TEXTURE 1,10,10
XROTATE OBJECT 1, -90

REM *** Create and texture cube ***
MAKE OBJECT CUBE 2,20
LOAD IMAGE "grid8by8.bmp",2
TEXTURE OBJECT 2,2
POSITION OBJECT 2,0,20,0

REM *** Create and texture sphere ***
MAKE OBJECT SPHERE 3,10,20,20
TEXTURE OBJECT 3,2
POSITION OBJECT 3, 20,10,30

REM *** Point the main light ***
SET POINT LIGHT 0,-10,50,0

REM *** Switch shading on for cube and sphere ***
SET SHADOW SHADING ON 2,-1,100,1
SET SHADOW SHADING ON 3,-1,100,1

REM *** Prepare objects for movement ***
POINT OBJECT 2,0,200,0
POINT OBJECT 3,20,10,-30
movement# = 0.1
DO
    REM *** Shadow on/off options ***
    IF INKEY$() = "s"
        SET GLOBAL SHADOWS ON
    ENDIF
    IF INKEY$() = "o"
        SET GLOBAL SHADOWS OFF
    ENDIF
    REM *** Rotate cube and move sphere ***
    TURN OBJECT LEFT 2,1
    MOVE OBJECT 2, 0.1
    MOVE OBJECT 3, movement#
    IF OBJECT POSITION Z(3) < -40
        movement# = -0.1
    ELSE
        IF OBJECT POSITION Z(3) > 30
            movement# = 0.1
        ENDIF
    ENDIF
    REM *** Display frame rate ***
    SET CURSOR 100,100
    PRINT "Frame rate : ",SCREEN FPS()
LOOP

REM *** End program ***
END
```

Activity 39.21

Type in and test the program in LISTING-39.11 (*advanced11.dbpro*). How does the frame rate change when shadows are switched off?

The SET GLOBAL SHADOW COLOR Statement

The colour of all shadows can be set using the SET GLOBAL SHADOW COLOR statement which has the format shown in FIG-39.32.

FIG-39.32 The SET GLOBAL SHADOW COLOR Statement



In the diagram:

<i>red</i>	is an integer value between 0 and 255 giving the strength of the red component within the shadow colour.
<i>green</i>	is an integer value between 0 and 255 giving the strength of the green component within the shadow colour.
<i>blue</i>	is an integer value between 0 and 255 giving the strength of the blue component within the shadow colour.
<i>alpha</i>	is an integer value between 0 and 255 giving the transparency component of the shadow. 0 - shadow invisible; 255 - shadow opaque.

For example, we could create a semi-transparent yellow shadow using the line:

```
SET GLOBAL SHADOW COLOR 255,255,0,100
```

Activity 39.22

Modify your last program so that the objects cast a semi-transparent red shadow.

The SET GLOBAL SHADOW SHADES Statement

In the last program, as the shadows of the cube and sphere overlapped, they merged, both shadows being the same shade. But in real life, when a second shadow moves over a first, it will often result in an even darker shadow where the two overlap. We can recreate this effect in our program using the SET GLOBAL SHADOW SHADES statement which has the format shown in FIG-39.33.

FIG-39.33
The SET GLOBAL SHADOW SHADES Statement



In the diagram:

<i>value</i>	is an integer value specifying how many variations in shadow shading are allowed
--------------	--

For example, if we knew three shadows might overlap and we wanted these overlaps to display different degrees of darkness, we would set *value* to 3.

Activity 39.23

In your last program, add a line before the DO..LOOP structure to allow 2 shadow shadings.

How does this affect the frame rate?

Positioning Shadows

Of course, a shadow's position is dependent on the position of the light creating that shadow; as we move the light, so the shadow will move.

Activity 39.24

In your last program, within the DO..LOOP structure, add the line:

```
POSITION LIGHT 0,LIGHT POSITION X(0)+1,100,0
```

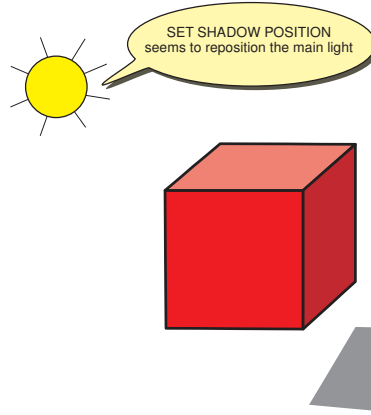
Run the program and observe how the shadow moves as the light changes position.

The SET SHADOW POSITION Statement

An alternative way of creating the same shifting shadow effect is to use the SET SHADOW POSITION statement. In practice, this statement seems to move light zero to a specified position (see FIG-39.34).

FIG-39.34

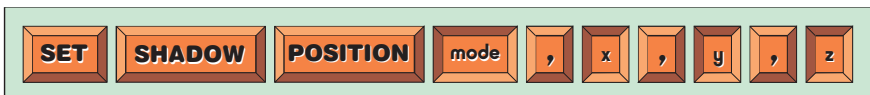
Positioning the Shadow



The SET SHADOW POSITION statement has the format shown in FIG-39.35.

FIG-39.35

The SET SHADOW POSITION Statement



In the diagram:

mode

has the value -1 (other values not known).

x,y,z

are real values giving the coordinates of the light responsible for creating the shadow.

For example, in our last program we could move the shadow to the right by specifying a light position to the left and above the cube with a line such as:

```
SET SHADOW POSITION -1,-65,60,0
```

Activity 39.25

In your last program, add the line

```
xpos = -100
```

immediately before the DO..LOOP structure.

Replace the POSITION LIGHT statement added in the last Activity by code to increment *xpos* and set the shadow position to the coordinates (*xpos*,60,0). How does this affect the shadows?

Shadows and Models

Although it is simple enough to create a shadow for a fundamental shape such as a cube or sphere, we have to work a little harder to create a realistic shadow for a model such as the Hivebrain creature we looked at back in Chapter 36.

LISTING-39.12 shows the shadow effect produced by the Hivebrain model when using the normal default settings for SET SHADOW SHADING ON.

LISTING-39.12

Model Shadows

```
REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
AUTOCAM OFF
POSITION CAMERA 0,10,-60
POINT CAMERA 0,0,0

REM *** Create background object ***
MAKE OBJECT PLAIN 1, 100,100
LOAD IMAGE "cobblestones.jpg",1
TEXTURE OBJECT 1,1
SCALE OBJECT TEXTURE 1,10,10
XROTATE OBJECT 1, -90

REM *** Load, scale and position alien ***
LOAD OBJECT "H-Alien Hivebrain-Move.x",2
SCALE OBJECT 2,300,300,300
POSITION OBJECT 2,0,0,0

REM *** Position light zero ***
SET POINT LIGHT 0,-60,50,0

REM *** Show on model's shadow ***
SET SHADOW SHADING ON 2,-1,100,1
REM *** End program ***
WAIT KEY
END
```

Activity 39.26

Type in and test the program in LISTING-39.12 (*advanced12.dbpro*).

As the program demonstrates, the shadow produced is far from realistic (see FIG-39.36).

FIG-39.36

The Default Shadow Cast
by a Complex Model.



To create a more convincing shadow, we need to turn the model into a mesh and then specify that mesh in the SET SHADOW SHADING ON statement. This requires the lines:

```
MAKE MESH FROM OBJECT 1,2  
SET SHADOW SHADING ON 2,1,100,1
```

Activity 39.27

Modify your last program as described and observe how the shadow changes.

As we can see in FIG-39.37, the shadow is now an exact match for the model.

FIG-39.37

An Improved Shadow



Our next problem arises when we play an animated model - the shadow stays fixed while the model itself changes. However, the shadow does at least move with the model.

Activity 39.28

Add the following code just before the REM *** End program *** statement:

```
REM *** Play model ***  
LOOP OBJECT 2
```

Activity 39.29

Change your program to finish with the lines:

```
REM *** Play model ***  
LOOP OBJECT 2  
  z# = 0  
  DO  
    z# = z# -0.1  
    POSITION OBJECT 2,0,0,z#  
  LOOP  
REM *** End program ***  
END
```

Check that the shadow moves with the model.

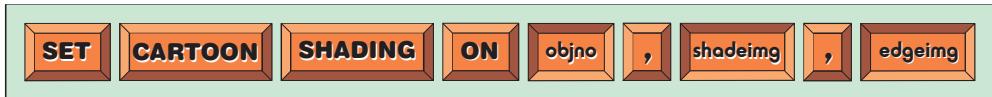
Other Shading Methods

The SET CARTOON SHADING ON Statement

A cartoon texturing effect can be achieved using the SET CARTOON SHADING ON statement which has the format shown in FIG-39.38.

FIG-39.38

The SET CARTOON SHADING ON Statement



In the diagram:

objno

is an integer value giving the ID of the object to which cartoon shading is to be applied.

shading

is an integer value giving the ID of the image to be used as the main texturing for the object.

edging

is an integer value giving the ID of the image to be used in edging the object texture.

Rather than texture a model in the normal way, cartoon shading colours the object with the main colour of the shade image and slightly modifies the overall effect using the *edge* image. Complex images have little impact on the final result, so the images used with this statement are usually small, simply coloured ones.

Using a black *edge* image will make the object completely black; using a white *edge* image allows the *shade* image to dominate in the final effect.

For example, we could use the images *shade.bmp* and *edge.bmp* to create a cartoon shading on the Hivebrain model using the following lines:

```
LOAD IMAGE "shade.bmp",1  
LOAD IMAGE "edge.bmp",2  
LOAD OBJECT "H-Alien Hivebrain-Move.x",1  
SET CARTOON SHADING ON 1,1,2
```


Activity 39.30

Write a program to add cartoon shading to a plane and the Hivebrain model used in the last program.

Try the program out with the following images:

shade1.bmp, edge1.bmp,
shade2.bmp, edge2.bmp,
shade3.bmp, edge3.bmp

FIG-39.39 shows several examples of the images used and the overall result when using cartoon shading on a plane and the Hivebrain model.

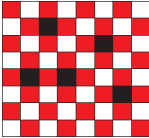
FIG-39.39

Examples of Cartoon Shading

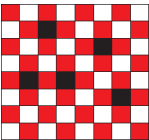
Shade Image

Edge Image

Result



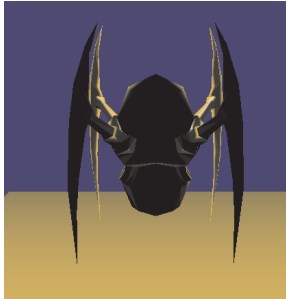
A single white pixel



A single black pixel



A single white pixel

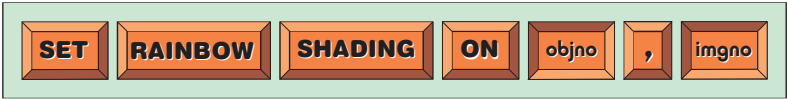


The SET RAINBOW SHADING ON Statement

The SET RAINBOW SHADING ON statement adds the colours of the rainbow to the surface of an object. The statement has the format shown in FIG-39.40.

FIG-39.40

The SET RAINBOW SHADING ON Statement



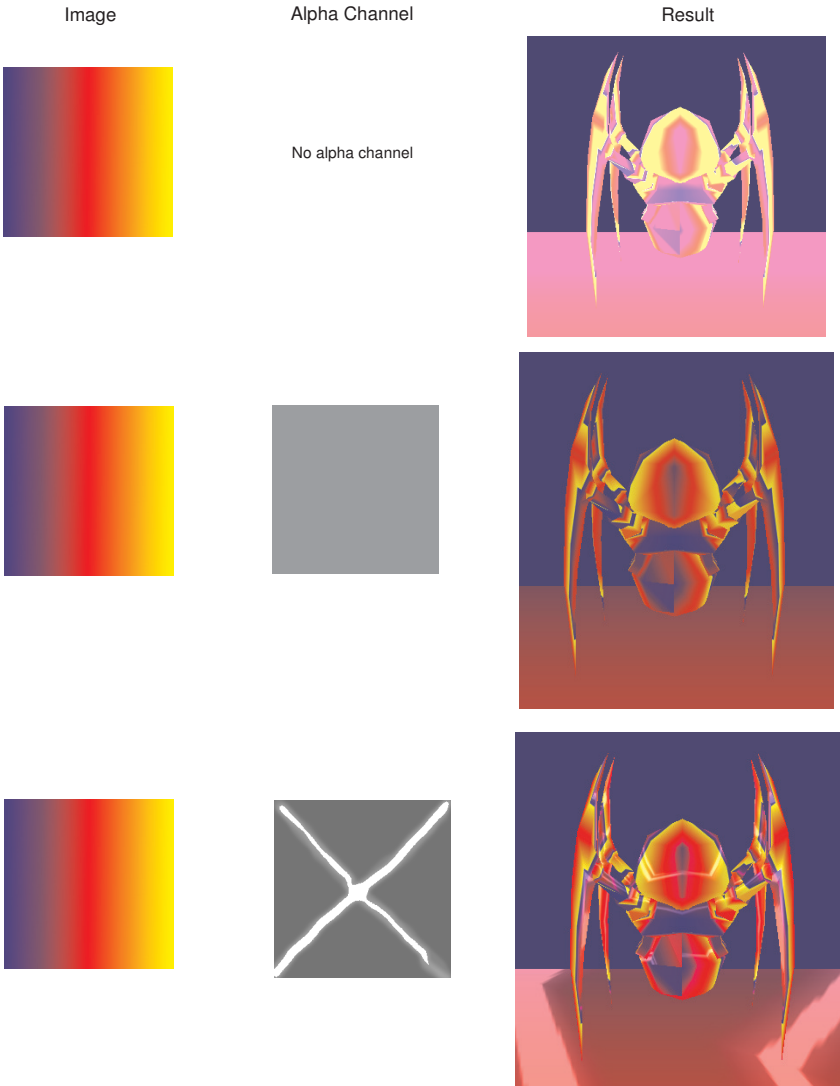
In the diagram:

- objno*
- is an integer value giving the ID of the object to which rainbow shading is to be applied.
- imgno*
- is an integer value giving the ID of the image to be used when creating the rainbow effect on the object.

The image used needs to have an alpha channel, otherwise the overall effect is too bright. FIG-39.41 shows the results achieved using an image with and without an alpha channel.

FIG-39.41

Rainbow Shading and Alpha Channels



Activity 39.31

Modify your last program to use rainbow shading instead of cartoon shading.

Test the program with the files *test1.tga*, *test2.tga* and *test3.tga* which correspond to the images used in FIG-39.41.

The SET REFLECTION SHADING ON Statement

In a previous chapter we used a second camera to create a mirror-like effect, but a more realistic way of creating a mirror-like surface is to use the SET REFLECTION SHADING ON statement. This will cause a specified object to reflect its surroundings, but the operation requires a significant amount of processing power and will only work if your video card is up to the job.

The SET REFLECTION SHADING ON statement has the format shown in FIG-39.42.

FIG-39.42

The SET REFLECTION SHADING ON Statement



In the diagram:

objno

is an integer value giving the ID of the object to which reflection shading is to be applied.

FIG-39.43 shows a plane reflecting cobbles and the Hivebrain model.

FIG-39.43

Reflections on a Plane



The program in LISTING-39.13 recreates the effect shown above.

LISTING-39.13

Reflections

```
ScreenSetUp ()
CreateSet ()
REM *** Create reflecting plane ***
MAKE OBJECT PLAIN 3,20,20
POSITION OBJECT 3,0,5,20
XROTATE OBJECT 3, 10
REM *** Rotate plane ***
SET REFLECTION SHADING ON 3
DO
    TURN OBJECT LEFT 3,0.5
LOOP
REM *** End Program ***
END
```

continued on next page

LISTING-39.13
(continued)

Reflections

```
FUNCTION ScreenSetUp()  
  SET DISPLAY MODE 1280,1024,32  
  COLOR BACKDROP RGB(255,255,100)  
  BACKDROP ON  
  AUTOCAM OFF  
  POSITION CAMERA 0,10,-30  
  POINT CAMERA 0,0,0  
ENDFUNCTION  
  
FUNCTION CreateSet()  
  REM *** Load images used ***  
  LOAD IMAGE "cobblestones.jpg",1  
  REM *** Create ground ***  
  MAKE OBJECT PLAIN 1,200,200  
  TEXTURE OBJECT 1,1  
  SCALE OBJECT TEXTURE 1,20,20  
  XROTATE OBJECT 1,-90  
  REM *** Load, scale and position alien ***  
  LOAD OBJECT "H-Alien Hivebrain-Move.x",2  
  SCALE OBJECT 2,300,300,300  
  POSITION OBJECT 2,0,0,0  
ENDFUNCTION
```

Activity 39.32

Type in and test the program in LISTING-39.13(*advanced13.dbpro*).

Do both sides of the plane reflect?

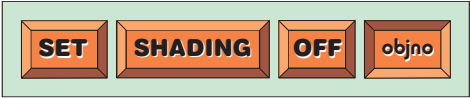
Modify the program to use a cube instead of a plane.

The SET SHADING OFF Statement

Any shading effect (shadow shading, rainbow shading, etc.) can be switched off using the SET SHADING OFF statement which has the format shown in FIG-39.44.

FIG-39.44

The SET SHADING OFF Statement



In the diagram:

objno is an integer value giving the ID of the object whose current shading effect is to be turned off.

Activity 39.33

Replace the reflective cube in your last program with a 20 by 20 plane and allow a key press to toggle the reflective ability of the plane.

Summary

- Different types of surfaces reflect light in different ways.
- Ambient reflection refers to how a surface reflects ambient light.
- Diffuse reflection is the scattered reflection of directed light.

- Specular reflection is the mirrored reflection of directed light.
- Use SET OBJECT AMBIENT to set the ambient reflection of an object.
- Use SET OBJECT DIFFUSE to set the diffuse reflection of an object.
- Use SET OBJECT SPECULAR to set the specular reflection of an object.
- Use SET OBJECT SPECULAR POWER to specify the degree of specular reflection from an object.
- Use SET OBJECT EMISSIVE to give the illusion that an object is emitting light.
- Use SET OBJECT LIGHT to specify how an object is to react to light.
- A light map is an image used to determine how an object reacts to light.
- Use SET LIGHT MAPPING ON to apply a light map to an object.
- A bump map is an image used to give the illusion of roughness on an object's surface.
- Use SET BUMP MAPPING ON to apply a bump map to an object.
- Use SET SPHERE MAPPING ON to create the illusion of an object's surroundings being reflected on its surface.
- Use SET BLEND MAPPING ON to blend two images on the surface of an object.
- Use SET CUBE MAPPING ON to create an object reflecting its surroundings.
- Use SET ALPHA MAPPING ON to make an object transparent or invisible.
- Use SET SHADOW SHADING ON to make an object cast a shadow.
- Use SET SHADOW POSITION to position the shadow cast by an object.
- Use SET SHADOW SHADING OFF to turn off the shadow effect for a specified object.
- Use SET GLOBAL SHADOWS to switch all allocated shadows on or off.
- Use SET GLOBAL SHADOW COLOR to set the colour-cast of all shadows.
- Use SET GLOBAL SHADOW SHADES to set the number of shades available when shadows overlap.
- To create a realistic shadow from a model, use the mesh of the model as the shadow.
- Use SET CARTOON SHADING ON to create a basic colouring of an object's surface.
- Use SET RAINBOW SHADING ON to create a rainbow effect on the surface of an object.
- Use SET REFLECTION SHADING ON to make an object's surface reflect the

objects around it.

- Use SET SHADING OFF to switch off all shading effects that have been applied to a specific object.

Solutions

Activity 39.1

```
REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP RGB(20,20,20)
BACKDROP ON
REM *** Programmed screen updates ***
SYNC ON
REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 0,0,-60
POINT CAMERA 0,0,0
REM *** Make cube ***
MAKE OBJECT SPHERE 1,10,50,50
REM *** Switch off directional light ***
HIDE LIGHT 0
SYNC
REM *** Switch off ambient reflection ***
PRINT "Change sphere's reflection of
↳ambient light"
PRINT "Press any key"
SYNC
WAIT KEY
SET OBJECT AMBIENT 1,0
SYNC
REM *** Switch on ambient reflection ***
PRINT "Change sphere's reflection of
↳ambient light"
PRINT "Press any key"
SYNC
WAIT KEY
SET OBJECT AMBIENT 1,1
SYNC
REM *** End program ***
WAIT KEY
END
```

Activity 39.2

```
REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP RGB(20,20,20)
BACKDROP ON
REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 0,0,-60
REM *** Create sphere ***
MAKE OBJECT SPHERE 1,10
COLOR OBJECT 1, RGB(0,0,255)
REM *** Switch off ambient reflection ***
WAIT KEY
SET OBJECT AMBIENT 1,0
REM *** Switch on diffuse reflection ***
WAIT KEY
SET OBJECT DIFFUSE 1, RGB(0,255,255)
REM *** End program ***
WAIT KEY
END
```

Activity 39.3

```
REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP RGB(20,20,20)
BACKDROP ON
REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 0,0,-60
REM *** Create sphere ***
MAKE OBJECT SPHERE 1,10
COLOR OBJECT 1, RGB(0,0,255)
```

```
REM *** Switch off ambient reflection ***
WAIT KEY
SET OBJECT AMBIENT 1,0
REM *** Switch on diffuse reflection ***
WAIT KEY
SET OBJECT DIFFUSE 1, RGB(0,255,255)
REM *** Add a yellow light ***
WAIT KEY
MAKE LIGHT 1
SET SPOT LIGHT 1,90,180
POSITION LIGHT 1, -1,3,-5
POINT LIGHT 1,0,0,0
COLOR LIGHT 1, RGB(255,255,0)
SET OBJECT SPECULAR 1, RGB(255,255,0)
REM *** End program ***
WAIT KEY
END
```

Activity 39.4

```
REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP RGB(20,20,20)
BACKDROP ON
REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 0,0,-60
REM *** Create sphere ***
MAKE OBJECT SPHERE 1,10
COLOR OBJECT 1, RGB(0,0,255)
REM *** Switch off ambient reflection ***
WAIT KEY
SET OBJECT AMBIENT 1,0
REM *** Switch on diffuse reflection ***
WAIT KEY
SET OBJECT DIFFUSE 1, RGB(0,255,255)
REM *** Add a yellow light ***
WAIT KEY
MAKE LIGHT 1
SET SPOT LIGHT 1,90,180
POSITION LIGHT 1, -1,3,-5
POINT LIGHT 1,0,0,0
COLOR LIGHT 1, RGB(255,255,0)
SET OBJECT SPECULAR 1, RGB(255,255,0)
FOR power = 100 TO 0 STEP -1
    SET OBJECT SPECULAR POWER 1,power
    WAIT 25
NEXT power
REM *** End program ***
WAIT KEY
END
```

Activity 39.5

No solution required.

Activity 39.6

```
REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP RGB(20,20,20)
BACKDROP ON
REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 0,0,-60
REM *** Create sphere ***
MAKE OBJECT SPHERE 1,10,100,100
REM *** Switch off lights ***
WAIT KEY
SET AMBIENT LIGHT 0
HIDE LIGHT 0
```

```

REM *** produce fake emissive effect ***
WAIT KEY
SET OBJECT EMISSIVE 1,RGB(0,0,255)
REM *** make cube ***
WAIT KEY
MAKE OBJECT CUBE 2,10
POSITION OBJECT 2,15,15,20
REM *** End program ***
WAIT KEY
END

```

The cube is not affected by the blue sphere and remains black.

```

WAIT KEY
MAKE LIGHT 1
SET POINT LIGHT 1,0,0,0
COLOR LIGHT 1, RGB(0,0,255)
REM *** Switch off cube reflection ***
SET OBJECT LIGHT 2,0
REM *** End program ***
WAIT KEY
END

```

This change causes the cube's surface to reflect a white light.

To change the ambient colour to red, modify the line

```
SET AMBIENT LIGHT 0
```

to read

```
SET AMBIENT LIGHT RGB(255,0,0)
```

This change makes the cube reflect the red ambient light until SET OBJECT LIGHT statement for the cube is executed. The cube's surface then returns to reflecting white light.

Activity 39.7

```

REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP RGB(20,20,20)
BACKDROP ON
REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 0,0,-60
REM *** Create sphere ***
MAKE OBJECT SPHERE 1,10,100,100
REM *** Switch off lights ***
WAIT KEY
SET AMBIENT LIGHT 0
HIDE LIGHT 0
REM *** produce fake emissive effect ***
WAIT KEY
SET OBJECT EMISSIVE 1,RGB(0,0,255)
REM *** make cube ***
WAIT KEY
MAKE OBJECT CUBE 2,10
POSITION OBJECT 2,15,15,20
REM *** Create light ***
WAIT KEY
MAKE LIGHT 1
SET POINT LIGHT 1,0,0,0
COLOR LIGHT 1, RGB(0,0,255)
REM *** End program ***
WAIT KEY
END

```

The cube is lit by the new light.

Activity 39.10

```

REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP RGB(20,20,20)
BACKDROP ON
REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 20,-4,-5
POINT CAMERA 0,0,3
CreateScene()
Rem *** End program ***
WAIT KEY
END

```

```

FUNCTION CreateScene()
    REM *** Create cobbled street ***
    MAKE OBJECT PLAIN 1,30,30
    XROTATE OBJECT 1,90
    LOAD IMAGE "cobblelarge.jpg",1
    TEXTURE OBJECT 1,1
    POSITION OBJECT 1,0,-10,0
    REM *** Create wall ***
    MAKE OBJECT PLAIN 2,30,20
    LOAD IMAGE "brickslarge.jpg",2
    TEXTURE OBJECT 2,2
    POSITION OBJECT 2,0,-1,5
    REM *** Create lamppost ***
    MAKE OBJECT CYLINDER 3,10
    SCALE OBJECT 3, 5,100,5
    POSITION OBJECT 3,0,-5,0
    LOAD IMAGE "lamppost.jpg",3
    TEXTURE OBJECT 3,3
    MAKE OBJECT SPHERE 4,2,100,100
    REM *** Make sphere appear to glow ***
    SET OBJECT EMISSIVE 4,RGB(255,255,255)
    REM *** Put light in sphere ***
    MAKE LIGHT 1
    SET POINT LIGHT 1,0,0,0
ENDFUNCTION

```

Activity 39.8

No solution required.

Activity 39.9

```

REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP RGB(20,20,20)
BACKDROP ON
REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 0,0,-60
REM *** Create sphere ***
MAKE OBJECT SPHERE 1,10,100,100
REM *** Switch off lights ***
WAIT KEY
SET AMBIENT LIGHT 0
HIDE LIGHT 0
REM *** produce fake emissive effect ***
WAIT KEY
SET OBJECT EMISSIVE 1,RGB(0,0,255)
REM *** make cube ***
WAIT KEY
MAKE OBJECT CUBE 2,10
POSITION OBJECT 2,15,15,20
REM *** Create light ***

```

Activity 39.11

To use the new light map change the line

```
LOAD IMAGE "lightmap.jpg",4
```

to


```
LOAD IMAGE "lightgrey.jpg", 4
```

This gives a less realistic lighting effect.

Activity 39.12

The main section of the program should read:

```
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP RGB(20,20,20)
BACKDROP ON
AUTOCAM OFF
POSITION CAMERA 20,-4,-5
POINT CAMERA 0,0,3
CreateScene()
LOAD IMAGE "lightgrey.jpg", 4
SET LIGHT MAPPING ON 2,4
SET LIGHT MAPPING ON 1,4
LOAD IMAGE "lampmap.jpg", 5
SET LIGHT MAPPING ON 3,5
REM *** Add bump map ***
WAIT KEY
LOAD IMAGE "cobblebump.jpg", 6
SET BUMP MAPPING ON 1,6
WAIT KEY
END
```

Unfortunately, adding bump mapping appears to remove the light mapping effect.

Activity 39.13

```
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP RGB(155,155,0)
BACKDROP ON
AUTOCAM OFF
POSITION CAMERA 0,10,-60
POINT CAMERA 0,0,0
REM *** make sphere ***
MAKE OBJECT SPHERE 1,10,50,50
WAIT KEY
REM *** Load and apply sphere map ***
LOAD IMAGE "fish.jpg", 1
SET SPHERE MAPPING ON 1,1
WAIT KEY
REM *** Texture sphere ***
LOAD IMAGE "stripes.jpg", 2
TEXTURE OBJECT 1,2
REM *** Rotate sphere ***
DO
    TURN OBJECT LEFT 1,1
    WAIT 10
LOOP
REM *** End program ***
END
```

While the texture rotates with the cube, the sphere mapped image remains stationary.

Activity 39.14

No solution required.

Activity 39.15

No solution required.

Activity 39.16

```
REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
AUTOCAM OFF
```

```
POSITION CAMERA 0,0,-50
POINT CAMERA 0,0,0
REM *** Add extra light ***
MAKE LIGHT 1
POSITION LIGHT 1,20,-20,50
REM *** Make cone with culling off ***
MAKE OBJECT CONE 22,10
SET OBJECT CULL 22,0
REM *** Load and apply sphere map ***
LOAD IMAGE "sky01.jpg", 1
LOAD IMAGE "sky02.jpg", 2
LOAD IMAGE "sky03.jpg", 3
LOAD IMAGE "cobblestones.jpg", 4
LOAD IMAGE "sky05.jpg", 5
LOAD IMAGE "sky06.jpg", 6
SET CUBE MAPPING ON 22,1,2,3,4,5,6
DO
    TURN OBJECT LEFT 22,RND(200)/100.0
    PITCH OBJECT UP 22,RND(200)/100.0
    ROLL OBJECT RIGHT 22,RND(200)/100.0 - 1
    WAIT 10
LOOP
REM *** End program ***
END
```

Activity 39.17

No solution required.

Activity 39.18

```
REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
AUTOCAM OFF
POSITION CAMERA 0,10,-60
POINT CAMERA 0,0,0
REM *** Create background object ***
MAKE OBJECT PLAIN 1,100,100
LOAD IMAGE "cobblestones.jpg", 1
TEXTURE OBJECT 1,1
SCALE OBJECT TEXTURE 1,10,10
XROTATE OBJECT 1,-90
REM *** Create and texture cube ***
MAKE OBJECT CUBE 2,20
LOAD IMAGE "grid8by8.bmp", 2
TEXTURE OBJECT 2,2
POSITION OBJECT 2,0,20,0
REM *** Adjust light ***
SET POINT LIGHT 0,-10,100,1
REM *** Switch cube's shadow on ***
SET SHADOW SHADING ON 2,-1,100,1
REM *** Rotate cube & display frame rate ***
DO
    TURN OBJECT LEFT 2,1
    SET CURSOR 100,100
    PRINT "Frame rate : ",SCREEN FPS()," "
    ↵,ord
LOOP
REM *** End program ***
END
```

The effect of the change on the author's machine was to slightly decrease the frame rate but stop a slight flicker effect on the side of the cube.

Changing the distance parameter to 10 removes the shadow since the distance from the cube to the floor plane is more than 10 units.

Activity 39.19

```
REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
AUTOCAM OFF
```

```

POSITION CAMERA 0,10,-60
POINT CAMERA 0,0,0
REM *** Create background object ***
MAKE OBJECT PLAIN 1,100,100
LOAD IMAGE "cobblestones.jpg",1
TEXTURE OBJECT 1,1
SCALE OBJECT TEXTURE 1,10,10
XROTATE OBJECT 1,-90
REM *** Create and texture cube ***
MAKE OBJECT CUBE 2,20
LOAD IMAGE "grid8by8.bmp",2
TEXTURE OBJECT 2,2
POSITION OBJECT 2,0,20,0
REM *** Adjust light ***
SET POINT LIGHT 0,-10,100,1
REM *** create mesh ***
MAKE OBJECT SPHERE 3,30,50,50
MAKE MESH FROM OBJECT 1,3
DELETE OBJECT 3
REM *** Switch cube's shadow on ***
SET SHADOW SHADING ON 2,1,100,1
REM *** Rotate cube and display frame
rate ***
DO
    TURN OBJECT LEFT 2,1
    SET CURSOR 100,100
    PRINT "Frame rate : ",SCREEN FPS()," ",
        ↵ord
LOOP
REM *** End program ***
END

```

The cube now casts a circular shadow! The frame rate is also reduced.

Activity 39.20

The DO..LOOP structure of the program should be changed to read:

```

DO
    IF INKEY$() <> ""
        SET SHADOW SHADING OFF 2
    ENDIF
    TURN OBJECT LEFT 2,1
    SET CURSOR 100,100
    PRINT "Frame rate : ",SCREEN FPS()," ",
        ↵ord
LOOP

```

Activity 39.21

The frame rate increases significantly when shadows are turned off.

Activity 39.22

```

REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
AUTOCAM OFF
POSITION CAMERA 0,10,-60
POINT CAMERA 0,0,0
REM *** Create background object ***
MAKE OBJECT PLAIN 1, 100,100
LOAD IMAGE "cobblestones.jpg",1
TEXTURE OBJECT 1,1
SCALE OBJECT TEXTURE 1,10,10
XROTATE OBJECT 1, -90
REM *** Create and texture cube ***
MAKE OBJECT CUBE 2,20
LOAD IMAGE "grid8by8.bmp",2
TEXTURE OBJECT 2,2
POSITION OBJECT 2,0,20,0
REM *** Create and texture sphere ***

```

```

MAKE OBJECT SPHERE 3,10,20,20
TEXTURE OBJECT 3,2
POSITION OBJECT 3, 20,10,30
REM *** Point the main light ***
SET POINT LIGHT 0,-10,50,0
REM *** Switch shading on ***
SET SHADOW SHADING ON 2,-1,100,1
SET SHADOW SHADING ON 3,-1,100,1
REM *** Prepare objects for movement ***
POINT OBJECT 2,0,200,0
POINT OBJECT 3,20,10,-30
movement# = 0.1
REM *** Change shadow colour ***
SET GLOBAL SHADOW COLOR 255,0,0,100
DO
    REM *** Shadow on/off options ***
    IF INKEY$() = "s"
        SET GLOBAL SHADOWS ON
    ENDIF
    IF INKEY$() = "o"
        SET GLOBAL SHADOWS OFF
    ENDIF
    REM *** Rotate cube and move sphere ***
    TURN OBJECT LEFT 2,1
    MOVE OBJECT 2, 0.1
    MOVE OBJECT 3, movement#
    IF OBJECT POSITION Z(3) < -40
        movement# = -0.1
    ELSE
        IF OBJECT POSITION Z(3) > 30
            movement# = 0.1
        ENDIF
    ENDIF
    REM *** Display frame rate ***
    SET CURSOR 100,100
    PRINT "Frame rate : ",SCREEN FPS()
LOOP
REM *** End program ***
END

```

Activity 39.23

Replace the lines:

```

REM *** Change shadow colour ***
SET GLOBAL SHADOW COLOR 255,0,0,100

```

with:

```

REM *** Set shadow shading ***
SET GLOBAL SHADOW SHADES 2

```

The frame rate will probably fall. The more shades specified, the lower the frame rate.

Activity 39.24

The DO..LOOP code should now read:

```

DO
    POSITION LIGHT 0,LIGHT POSITION X(0)
    ↵+1,100,0
    REM *** Shadow on/off options ***
    IF INKEY$() = "s"
        SET GLOBAL SHADOWS ON
    ENDIF
    IF INKEY$() = "o"
        SET GLOBAL SHADOWS OFF
    ENDIF
    REM *** Rotate cube and move sphere ***
    TURN OBJECT LEFT 2,1
    MOVE OBJECT 2, 0.1
    MOVE OBJECT 3, movement#
    IF OBJECT POSITION Z(3) < -40

```

```

movement# = -0.1
ELSE
  IF OBJECT POSITION Z(3) > 30
    movement# = 0.1
  ENDIF
ENDIF
REM *** Display frame rate ***
SET CURSOR 100,100
PRINT "Frame rate : ",SCREEN FPS()
LOOP

```

Activity 39.25

```

xpos = -100
DO
  INC xpos
  SET SHADOW POSITION -1,xpos,60,0
  REM *** Shadow on/off options ***
  IF INKEY$() = "s"
    SET GLOBAL SHADOWS ON
  ENDIF
  IF INKEY$() = "o"
    SET GLOBAL SHADOWS OFF
  ENDIF
  REM *** Rotate cube and move sphere ***
  TURN OBJECT LEFT 2,1
  MOVE OBJECT 2, 0.1
  MOVE OBJECT 3, movement#
  IF OBJECT POSITION Z(3) < -40
    movement# = -0.1
  ELSE
    IF OBJECT POSITION Z(3) > 30
      movement# = 0.1
    ENDIF
  ENDIF
  REM *** Display frame rate ***
  SET CURSOR 100,100
  PRINT "Frame rate : ",SCREEN FPS()
LOOP

```

The shadow shifts as the light moves.

Activity 39.26

No solution required.

Activity 39.27

```

REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
AUTOCAM OFF
POSITION CAMERA 0,10,-60
POINT CAMERA 0,0,0
REM *** Create background object ***
MAKE OBJECT PLAIN 1, 100,100
LOAD IMAGE "cobblestones.jpg",1
TEXTURE OBJECT 1,1
SCALE OBJECT TEXTURE 1,10,10
XROTATE OBJECT 1,-90
REM *** Load, scale and position alien ***
LOAD OBJECT "H-Alien Hivebrain-Move.x",2
SCALE OBJECT 2,300,300,300
POSITION OBJECT 2,0,0,0
REM *** Position light zero ***
SET POINT LIGHT 0,-60,50,0
REM *** Show on model's shadow ***
MAKE MESH FROM OBJECT 1,2
SET SHADOW SHADING ON 2,1,100,1
REM *** End program ***
WAIT KEY
END

```

ACTIVITY 39.28

```

REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
AUTOCAM OFF
POSITION CAMERA 0,10,-60
POINT CAMERA 0,0,0
REM *** Create background object ***
MAKE OBJECT PLAIN 1, 100,100
LOAD IMAGE "cobblestones.jpg",1
TEXTURE OBJECT 1,1
SCALE OBJECT TEXTURE 1,10,10
XROTATE OBJECT 1,-90
REM *** Load, scale and position alien ***
LOAD OBJECT "H-Alien Hivebrain-Move.x",2
SCALE OBJECT 2,300,300,300
POSITION OBJECT 2,0,0,0
REM *** Position light zero ***
SET POINT LIGHT 0,-60,50,0
REM *** Show on model's shadow ***
MAKE MESH FROM OBJECT 1,2
SET SHADOW SHADING ON 2,1,100,1
REM *** Play model ***
LOOP OBJECT 2
REM *** End program ***
WAIT KEY
END

```

Activity 39.29

```

REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
AUTOCAM OFF
POSITION CAMERA 0,10,-60
POINT CAMERA 0,0,0
REM *** Create background object ***
MAKE OBJECT PLAIN 1, 100,100
LOAD IMAGE "cobblestones.jpg",1
TEXTURE OBJECT 1,1
SCALE OBJECT TEXTURE 1,10,10
XROTATE OBJECT 1, -90
REM *** Load, scale and position alien ***
LOAD OBJECT "H-Alien Hivebrain-Move.x",2
SCALE OBJECT 2,300,300,300
POSITION OBJECT 2,0,0,0
REM *** Position light zero ***
SET POINT LIGHT 0,-60,50,0
REM *** Show on model's shadow ***
MAKE MESH FROM OBJECT 1,2
SET SHADOW SHADING ON 2,1,100,1
REM *** Play model ***
LOOP OBJECT 2
z# = 0
DO
  z# = z# -0.1
  POSITION OBJECT 2,0,0,z#
LOOP
REM *** End program ***
WAIT KEY
END

```

Activity 39.30

```

REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP RGB(255,255,100)
BACKDROP ON
AUTOCAM OFF
POSITION CAMERA 0,10,-30
POINT CAMERA 0,0,0|
REM *** Load images used ***
LOAD IMAGE "shade1.bmp",1
LOAD IMAGE "edge2.bmp",2
REM *** Create ground ***
MAKE OBJECT PLAIN 1,200,200
XROTATE OBJECT 1,-90

```

```

REM *** Load, scale and position alien ***
LOAD OBJECT "H-Alien Hivebrain-Move.x",2
SCALE OBJECT 2,300,300,300
POSITION OBJECT 2,0,0,0
REM *** Use cartoon shading ***
SET CARTOON SHADING ON 1,1,2
SET CARTOON SHADING ON 2,1,2
REM *** End program ***
WAIT KEY
END

```

```

SCALE OBJECT 2,300,300,300
POSITION OBJECT 2,0,0,0
ENDFUNCTION

```

Activity 39.33

The main section is coded as:

```

REM *** Set up screen ***
ScreenSetUp()
CreateSet()
REM *** Create reflecting plane ***
MAKE OBJECT PLAIN 3,20,20
POSITION OBJECT 3,0,5,20
XROTATE OBJECT 3, 10
REM *** Rotate plane ***
SET REFLECTION SHADING ON 3
reflection = 1
DO
    TURN OBJECT LEFT 3,0.5
    ch$ = INKEY$()
    IF ch$ <> ""
        IF reflection = 1
            SET SHADING OFF 3
            reflection = 0
        ELSE
            SET REFLECTION SHADING ON 3
            reflection = 1
        ENDIF
    REM *** Wait for key release ***
    WHILE INKEY$() <> ""
        TURN OBJECT LEFT 3,0.5
    ENDWHILE
ENDIF
LOOP
REM *** End Program ***
END

```

Activity 39.31

```

REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP RGB(255,255,100)
BACKDROP ON
AUTOCAM OFF
POSITION CAMERA 0,10,-30
POINT CAMERA 0,0,0
REM *** Load image used ***
LOAD IMAGE "test3.tga",1
REM *** Create ground ***
MAKE OBJECT PLAIN 1,200,200
XROTATE OBJECT 1,-90
REM *** Load, scale and position alien ***
LOAD OBJECT "H-Alien Hivebrain-Move.x",2
SCALE OBJECT 2,300,300,300
POSITION OBJECT 2,0,0,0
REM *** Use rainbow shading ***
SET RAINBOW SHADING ON 1,1
SET RAINBOW SHADING ON 2,1
REM *** End program ***
WAIT KEY
END

```

Activity 39.32

Only one side of the plane is reflective.

```

REM *** Set up screen ***
ScreenSetUp()
CreateSet()
REM *** Create reflecting cube ***
MAKE OBJECT CUBE 3,20
POSITION OBJECT 3,0,5,20
XROTATE OBJECT 3, 10
REM *** Rotate plane ***
SET REFLECTION SHADING ON 3
DO
    TURN OBJECT LEFT 3,0.5
LOOP
REM *** End Program ***
END

FUNCTION ScreenSetUp()
    SET DISPLAY MODE 1280,1024,32
    COLOR BACKDROP RGB(255,255,100)
    BACKDROP ON
    AUTOCAM OFF
    POSITION CAMERA 0,10,-30
    POINT CAMERA 0,0,0
ENDFUNCTION

FUNCTION CreateSet()
    REM *** Load images used ***
    LOAD IMAGE "cobblestones.jpg",1
    LOAD IMAGE "test3.tga",2
    REM *** Create ground ***
    MAKE OBJECT PLAIN 1, 200,200
    TEXTURE OBJECT 1,1
    SCALE OBJECT TEXTURE 1,20,20
    XROTATE OBJECT 1,-90
    REM *** Load, scale and place alien ***
    LOAD OBJECT "H-Alien Hivebrain-Move.x"
    2,2

```

Bounding Boxes and Spheres

Object Collisions

Sliding Collisions

Static Collisions

Object Collisions

Introduction

Back in Chapter 21 we saw the importance of detecting when two sprites collide - for example, we needed to know when a bat hit a ball or a missile hit a spaceship. That same requirement exists when dealing with a 3D world; we need to detect a sphere's collision with a cube or a bullet with a wall.

Unfortunately, things aren't as simple in 3D as they are in 2D, and this is reflected by the number and complexity of the commands available for 3D collision handling.

There are two main classes of collision in 3D. The first, **object collision**, deals with collisions between moveable 3D objects (such as spheres, cones, cubes, and loaded models). The second type is **static collision** where a moving object collides with a fixed (static) object (for example, a character colliding with the side of a house).

Moveable objects are often referred to as **dynamic** objects.

Object Collision

When one 3D object collides with another, we need to detect this and make the objects react to the event in a realistic way.

The program in LISTING-40.1 moves a sphere towards a cube.

LISTING-40.1

When 3D Objects Collide

```
REM *** Setup screen ***
SET DISPLAY MODE 1280, 1024, 32
COLOR BACKDROP 0
BACKDROP ON

REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 100,10,-200
POINT CAMERA 0,0,0

REM *** Make two objects ***
MAKE OBJECT CUBE 1, 40
MAKE OBJECT SPHERE 2, 5
POSITION OBJECT 2, 0,0,-100

REM *** Move sphere towards cube ***
DO
    POSITION OBJECT 2,0,0,OBJECT POSITION Z(2)+1
LOOP

REM *** End program ***
END
```

Activity 40.1

Type in and test the program given in LISTING-40.1 (*collision01.dbpro*).

What happens when the sphere comes into contact with the cube?

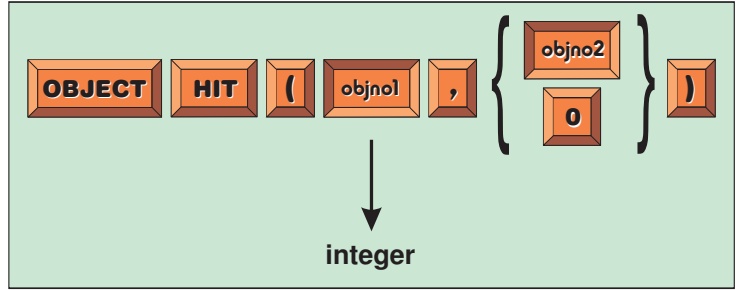
As we saw from the Activity above, objects will not automatically react to making contact with each other. If we want some reaction, then we must include explicit code to deal with this in our program.

The OBJECT HIT Statement

We can detect if two objects have collided using the OBJECT HIT statement which has the format given in FIG-40.1.

FIG-40.1

The OBJECT HIT Statement



In the diagram:

- | | |
|---------------|---|
| <i>objno1</i> | is an integer value identifying the main object being tested. |
| <i>objno2</i> | is an integer value identifying the second object to be tested. |
| 0 | use this option (zero) when the ID of the second object is unknown. |

When using the first option, where both objects are specified using ID numbers, the statement returns 1 if the objects have collideded and zero if no collision has taken place. For example, in the last program we could check for the sphere and cube colliding using the statement:

```
IF OBJECT HIT(1,2) = 1
```

When we are unsure of which object may have been hit, we can use the second version of this statement (with zero as the final value). This will return the ID number of any object struck by *objno1*. If no object has been hit, then zero is returned. For example, we could discover the ID of any object struck by object 1 with the code:

```
objectstruck = OBJECT HIT(1,0)
IF objectstruck = 0
  CURSOR 100,100
  PRINT "No object hit"
ELSE
  CURSOR 100,100
  PRINT "Object hit has ID ",objectstruck
ENDIF
```

Activity 40.2

Modify your last program (*collision01.dbpro*) so that the DO..LOOP structure is exited when the sphere collides with the cube.

Add a second DO..LOOP structure near the end of the program to allow the player to move the camera using the arrowkeys.

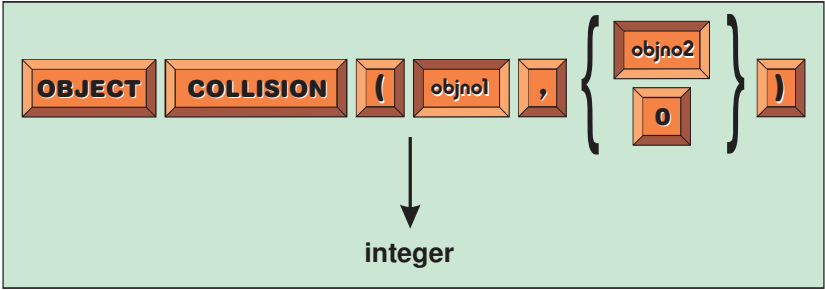
Run the modified version of the program and manually position the camera so that you can see that the sphere has come into contact with the cube.

The OBJECT COLLISION Statement

An alternative to OBJECT HIT is OBJECT COLLISION. The two statements perform exactly the same function. This statement has the format shown in FIG-40.2.

FIG-40.2

The OBJECT COLLISION Statement



In the diagram:

- objno1* is an integer value identifying the main object being tested.
- objno2* is an integer value identifying the second object to be tested.
- 0* use this option (zero) when the ID of the second object is unknown.

Activity 40.3

Modify your last program to use the OBJECT COLLISION statement in place of OBJECT HIT.

The SET OBJECT COLLISION Statement

It is possible to switch off collision detection for a specific object using the SET OBJECT COLLISION statement. The same statement can be used to switch collision detection back on. The statement has the format shown in FIG-40.3.

FIG-40.3

The SET OBJECT COLLISION Statement



In the diagram:

- objno* is an integer value identifying the object whose collision detection is to be modified.

When an object has its collision detection switched off using this statement, any subsequent calls to OBJECT HIT or OBJECT COLLISION which involve this object will always return zero.

Activity 40.4

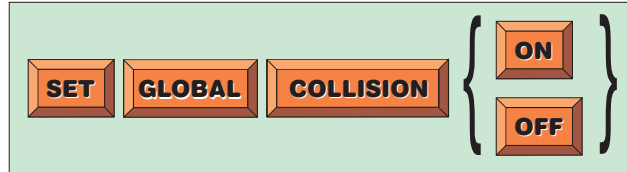
Switch off collision detection for object 1 in your last program and check out how this affects the results obtained.

The SET GLOBAL COLLISION Statement

While SET OBJECT COLLISION allows us to control collision detection for an individual object, we can control collision detection of all 3D objects using the SET GLOBAL COLLISION statement. This statement allows every object's collision detection to be switched on or off, irrespective of any previous setting assigned to individual objects. The SET GLOBAL COLLISION statement has the format shown in FIG-40.4.

FIG-40.4

The SET GLOBAL COLLISION Statement



NOTE: SET GLOBAL COLLISION ON does not neutralise the SET OBJECT COLLISION OFF 1 statement! (DBPro v 1.062)

Activity 40.5

Using the SET GLOBAL COLLISION statement, neutralise the effect of the SET OBJECT COLLISION statement added to your program in the previous Activity.

How Collision Detection Works

DarkBASIC Pro automatically places an invisible box around every 3D object that appears on the screen. This box is known as a **bounding box** or a **collision box**. When the collision boxes of two shapes come into contact with each other, a collision is detected.

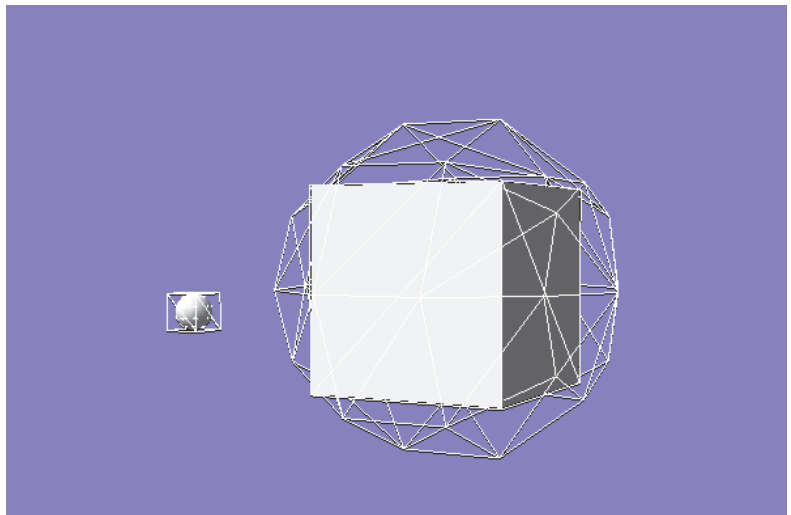
The bounding sphere is really a polyhedron.

In addition to the collision box, every object is also surrounded by a **bounding** or **collision sphere** since, for some 3D objects, this creates a more accurate approximation of the shape it surrounds than a box. For example, FIG-40.5 shows the box and sphere that surround each of the objects in our previous program.

FIG-40.5

Collision Boxes

Since they are almost perfect matches, the 3D sphere's collision sphere and the 3D cube's collision box are difficult to spot in the screen dump.



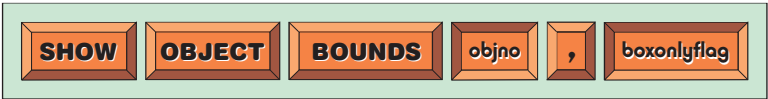
Notice that the collision box is a perfect match for the cube object, while the bounding sphere more accurately specifies the volume of the sphere object.

The SHOW OBJECT BOUNDS Statement

It is possible to make the bounding box and sphere that surround a specific 3D object visible using the SHOW OBJECT BOUNDS statement which has the format shown in FIG-40.6.

FIG-40.6

The SHOW OBJECT BOUNDS Statement



In the diagram:

objno

is an integer value specifying the object whose bounding volumes are to be displayed.

boxonlyflag

0 or 1. Set to 0 to show both the box and sphere bounding volumes; set to 1 to show only the bounding box.

Activity 40.6

Modify your last program so that the bounding boxes of both objects are displayed.

Activity 40.7

Change the cube in your program to a cone. How does this change affect the collision between the two objects?

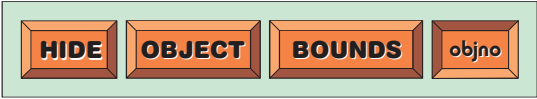
This last Activity highlights a problem: boxes and spheres do not make ideal collision volumes for every shape.

The HIDE OBJECT BOUNDS statement

An object's bounding box and sphere can be hidden again using the HIDE OBJECT BOUNDS statement which has the format shown in FIG-40.7.

FIG-40.7

The HIDE OBJECT BOUNDS Statement



In the diagram:

objno

is an integer value specifying the object whose bounding volumes are to be hidden.

Activity 40.8

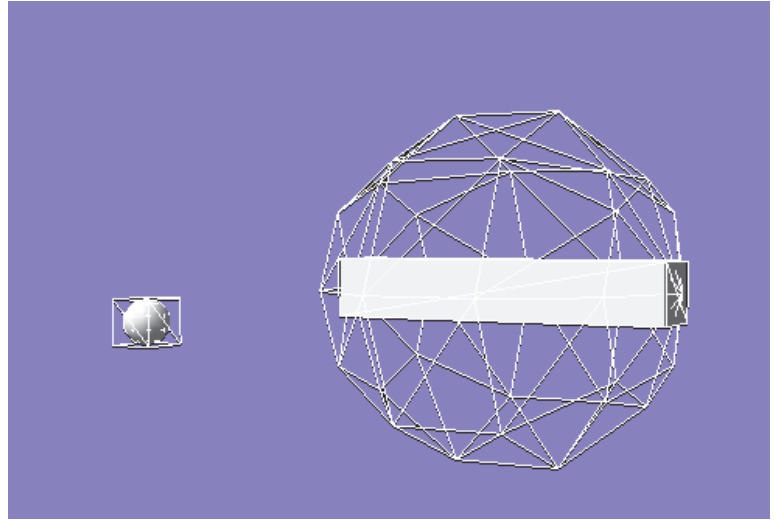
In the last program, use HIDE OBJECT BOUNDS to hide the bounding boxes of the sphere and cone after a key is pressed and before movement starts.

We're using the term **cuboid** to refer to the 3D box object and the term **box** to refer to its surrounding bounding box.

If a 3D object is moved, its bounding volumes (box and sphere) move with it. If a 3D object is rotated, its bounding box also rotates, but the bounding sphere, being symmetrical, does not need to rotate. FIG-40.8 shows the bounding volumes for a sphere and cuboid after the cuboid has been rotated by 90° about the z-axis. From this we can see that the cuboid's bounding box has also been rotated.

FIG-40.8

A Collision Box Rotates Along with the Object to which it is Attached



By default, the collision boxes are always used to detect collision between two 3D objects - even between two spheres. The program in LISTING-40.2 fires the sphere in the direction of the rotated cuboid shown in FIG-40.8 above. Since the objects' collision boxes do not intersect, no collision is detected.

LISTING-40.2

Only a Hit Between Two Collision Boxes is Detected

```
REM *** Setup screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP 0
BACKDROP ON

REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 100,10,-200
POINT CAMERA 0,0,0

REM *** Make box ***
MAKE OBJECT BOX 1,10,60,10
REM *** Display its bounding volumes ***
SHOW OBJECT BOUNDS 1
REM *** Lie box on its side ***
ZROTATE OBJECT 1, 90

REM *** Create a sphere ***
MAKE OBJECT SPHERE 2, 5
SHOW OBJECT BOUNDS 2
REM *** Position sphere ***
POSITION OBJECT 2,0,20,-100

REM *** Move sphere towards box until they collide ***
DO
  POSITION OBJECT 2,0,20,OBJECT POSITION Z(2)+1
  IF OBJECT HIT(1,2)
    EXIT
  ENDIF
LOOP
REM *** End program ***
WAIT KEY
END
```

Activity 40.9

Type in and test the program given in LISTING-40.2 (*collision02.dbpro*).

We can see from running this program that, although the bounding spheres of both 3D objects intersect, no collision is detected.

Modifying Collision Detection

The SET OBJECT COLLISION TO SPHERES Statement

It is possible to use the bounding spheres instead of the bounding boxes to detect collision by using the SET OBJECT COLLISION TO SPHERES statement which has the format shown in FIG-40.9.

FIG-40.9

The SET OBJECT
COLLISION TO
SPHERES Statement



In the diagram:

objno

is an integer value specifying the ID of the object which is to use its bounding sphere when detecting collisions.

Activity 40.10

Add the line

```
SET OBJECT COLLISION TO SPHERES 1
```

to your last program immediately after the cuboid object is created.

How does this change affect the program?

The SET OBJECT RADIUS Statement

When using a collision sphere, it is possible to modify the radius of that sphere from its default size (based on the size of the object it surrounds) using the SET OBJECT RADIUS statement, which has the format shown in FIG-40.10.

FIG-40.10

The SET OBJECT
RADIUS Statement



In the diagram:

objno

is an integer value specifying the object whose bounding sphere's radius is to be modified.

radius

is a real number giving the radius of the collision sphere.

Activity 40.11

Move the sphere in your previous program to a height of 40 and run the code. Does the 3D sphere hit the cuboid's bounding sphere? Change the radius of the cuboid's bounding sphere to 50 and re-run the program. Does the 3D sphere collide with the modified bounding sphere?

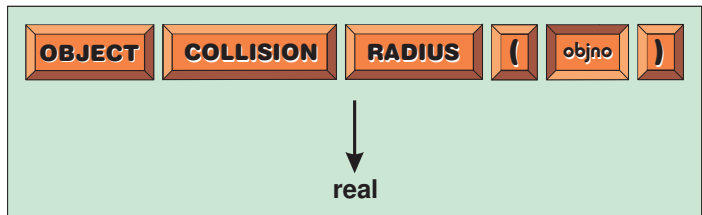
Unfortunately, the SHOW OBJECT BOUNDS statement only displays the default bounding box and sphere, so it does not show any changes to the bounding sphere's size when the radius is changed.

The OBJECT COLLISION RADIUS Statement

If we need to find out the current radius of a collision sphere we can use the OBJECT COLLISION RADIUS statement, the format for which is shown in FIG-40.11.

FIG-40.11

The OBJECT COLLISION RADIUS Statement



In the diagram:

objno

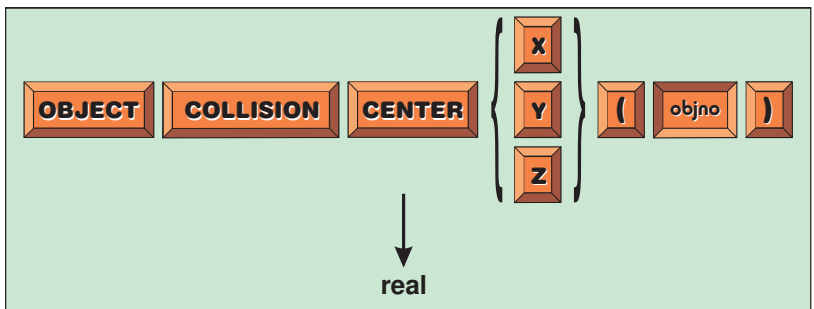
is an integer value specifying the object whose collision sphere's radius is to be determined. This object should previously have been set to use a collision sphere.

The OBJECT COLLISION CENTER Statement

The centre of a collision sphere can be determined using the OBJECT COLLISION CENTER statement which has the format shown in FIG-40.12.

FIG-40.12

The OBJECT COLLISION CENTER Statement



In the diagram:

X,Y,Z

Use the appropriate one of these options for the ordinate required.

objno

is an integer value specifying the object whose collision sphere's centre is to be determined. This object should previously have been set to use a collision sphere.

For example, if object 2 employs a collision sphere, then we can find that sphere's centre with the lines:

```
x# = OBJECT COLLISION CENTER X (2)
y# = OBJECT COLLISION CENTER Y (2)
z# = OBJECT COLLISION CENTER Z (2)
```

The SET OBJECT COLLISION TO BOXES Statement

It is possible to revert to using an object's collision box with the SET OBJECT COLLISION TO BOXES statement which has the format shown in FIG-40.13.

FIG-40.13

The SET OBJECT
COLLISION TO BOXES
Statement



In the diagram:

objno

is an integer value specifying the object which is to use its bounding box when detecting collisions.

The SET OBJECT COLLISION TO POLYGONS Statement

There are some shapes for which boxes and spheres will just not be accurate enough when trying to determine if a collision has occurred.

Activity 40.12

Reload your earlier program (*collision01.dbpro*) and check the difference between using a box or a sphere to detect collision between the 3D sphere and cone.

We can force the computer to detect collision with the actual polygons that make up the 3D shape itself by using the SET OBJECT COLLISION TO POLYGONS statement, the format for which is shown in FIG-40.14.

FIG-40.14

The SET OBJECT
COLLISION TO
POLYGONS Statement



In the diagram:

objno

is an integer value specifying the object which is to use its own polygons when detecting collisions.

In LISTING-40.3 the program again fires the sphere towards the cone shape, but this time using polygon collision detection.

LISTING-40.3

Using Polygon Collision
Detection

```
REM *** Setup screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP 0
BACKDROP ON

REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA -100,10,-20
POINT CAMERA 0,0,0
```

continued on next page

LISTING-40.3

(continued)

Using Polygon Collision
Detection

```
REM *** Make cone ***
MAKE OBJECT CONE 1,40
REM *** Use polygon collision detection ***
SET OBJECT COLLISION TO POLYGONS 1

REM *** Create and position a sphere ***
MAKE OBJECT SPHERE 2, 5
POSITION OBJECT 2,0,10,-100

REM *** Move sphere towards cone until they collide ***
DO
  POSITION OBJECT 2,0,10,OBJECT POSITION Z(2)+1
  IF OBJECT HIT(1,2)
    EXIT
  ENDIF
LOOP

REM *** End program ***
WAIT KEY
END
```

Activity 40.13

Type in and test the program given in LISTING-40.3 (*collision03.dbpro*).

Does the sphere come into contact with the cone?

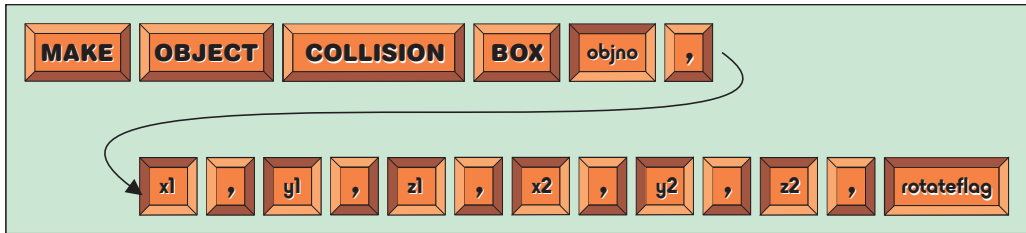
Using polygon collision detection is very process-intensive and can slow down your program, so don't use it unnecessarily.

The MAKE OBJECT COLLISION BOX Statement

Assuming we want to use a bounding box to detect collisions rather than spheres or polygons, it is possible to replace the default bounding box of an object with one of your own using the MAKE OBJECT COLLISION BOX statement. This statement has the format shown in FIG-40.15.

FIG-40.15

The MAKE OBJECT COLLISION BOX Statement



In the diagram:

objno

is an integer value specifying the object whose bounding volumes are to be displayed.

x1,y1,z1

are a set of real values specifying the near bottom-left vertex of the bounding box.

x2,y2,z2

are a set of real values specifying the far top-right vertex of the bounding box.

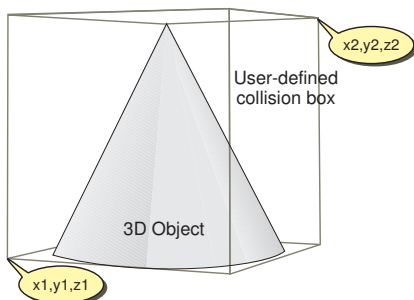
rotateflag

0 or 1. If set to 1, the bounding box will rotate when the object to which it is linked rotates; when set to zero the bounding box is unaffected by rotation of the object to which it is linked.

The diagram in FIG-40.16 shows the position of the vertices to be given when creating a collision box.

FIG-40.16

The Coordinates Required
When Specifying a Collision
Box



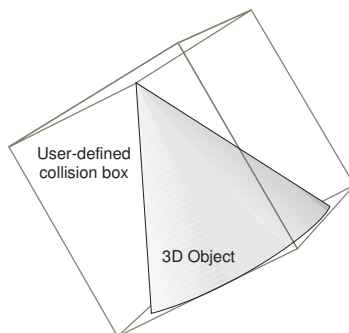
Normally, a collision box will rotate along with the 3D object to which it is attached. For example, FIG-40.17 shows the position of a cone (object 1) and its corresponding collision box after the statement

```
ZROTATE OBJECT 1, 45
```

has been executed.

FIG-40.17

A Collision Box Rotates
with the Object to which
it is Attached



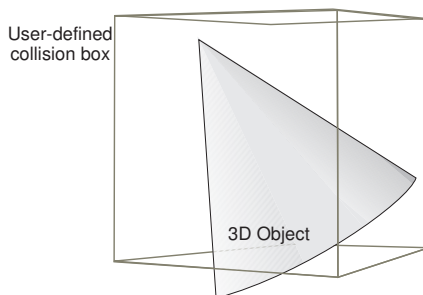
By using a setting of zero for *rotateflag*, the collision box will not rotate along with the object to which it is attached. For example, if we execute the lines

```
MAKE OBJECT CONE 1,40  
MAKE OBJECT COLLISION BOX 1,-20,-20,-20,20,20,20,0  
ZROTATE OBJECT 1,45
```

rotating the cone by 45°, then the cone and its collision box will be as shown in FIG-40.18.

FIG-40.18

The Collision Box Does
Not Rotate

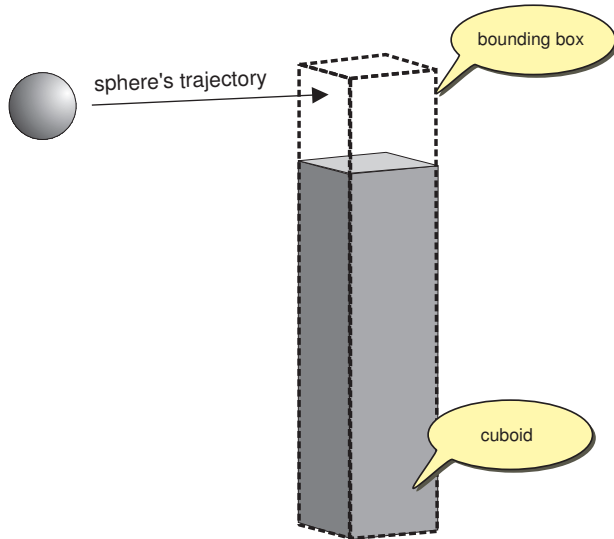


Although the collision box does not rotate, it will move should the object to which it is attached be repositioned.

Unfortunately, the SHOW OBJECT BOUNDS statement is not capable of showing a user-defined collision box, so we can't see the effects of creating a new collision box or preventing its rotation. However, we can observe how it affects collisions in the program given in LISTING-40.4. In this program a cuboid is given a bounding box which is taller than the object itself. As a sphere passes above the cuboid it collides with the bounding box and a collision is detected. FIG-40.19 shows how the program executes.

FIG-40.19

A Box Object and its
Collision Box



LISTING-40.4

Using MAKE OBJECT
COLLISION BOX

```
REM *** Setup screen ***
SET DISPLAY MODE 1280, 1024, 32
COLOR BACKDROP 0
BACKDROP ON

REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 100,10,-20
POINT CAMERA 0,0,0

REM *** Make box ***
MAKE OBJECT BOX 1, 10,40,10
REM *** Create a bounding box taller than 3D box ***
MAKE OBJECT COLLISION BOX 1,-5,-20,-5,5,40,5,0

REM *** Make and position sphere ***
MAKE OBJECT SPHERE 2, 5
POSITION OBJECT 2, 0,30,-100

REM *** Move sphere towards box until they collide ***
DO
  POSITION OBJECT 2,0,30,OBJECT POSITION Z(2)+1
  SET CURSOR 100,100
  PRINT OBJECT POSITION Z(2)," ",STATISTIC(1)
  IF OBJECT COLLISION(1,2)
    EXIT
  ENDIF
LOOP

REM *** End program ***
WAIT KEY
END
```

Activity 40.14

Type in and test the program in LISTING-40.4 (*collision04.dbpro*).

Does the sphere stop when it strikes the box?

Modify the program so that the cuboid is rotated to 90° about its z-axis after the collision box has been created.

How does this affect the sphere?

Change the final value in the MAKE OBJECT COLLISION BOX to 1.

How is the sphere affected by this change?

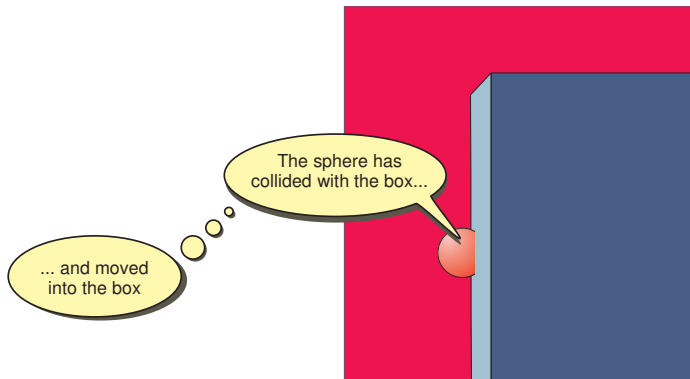
The first time the cuboid is rotated, the sphere still collides with its bounding box because the bounding box no longer rotates in conjunction with its object.

The GET OBJECT COLLISION Statement

When one object collides with a second object, and that second object has an object collision box (created using MAKE OBJECT COLLISION BOX), and the object collision box is designed not to rotate with the object to which it has been linked, then it is possible to determine how far object 1 has moved into the collision box of object 2 (see FIG-40.20).

FIG-40.20

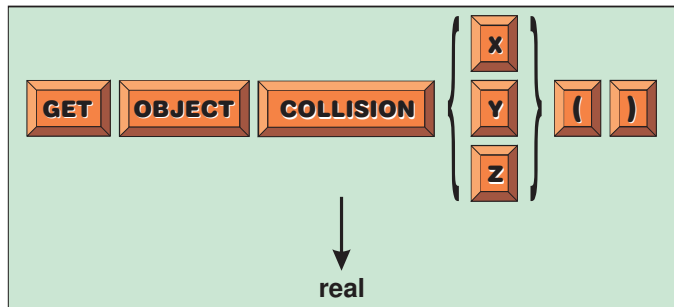
A Sphere Moving
Through a Box



The amount of penetration can be determined by the GET OBJECT COLLISION statement which has the format shown in FIG-40.21.

FIG-40.21

The GET OBJECT
COLLISION Statement



In the diagram:

X,Y,Z

Use one of these options to determine the penetration of the object within the collision box in the specified direction.

Normally, we would use all three options to get the full details of the collision:

```
REM *** Create a collision box round one object ***
MAKE OBJECT COLLISION BOX 1,-10,-20,-10,10,40,10,0
.
.
REM *** IF collision get penetration details ***
IF OBJECT COLLISION(1,2)
  x# = GET OBJECT COLLISION X()
  y# = GET OBJECT COLLISION Y()
  z# = GET OBJECT COLLISION Z()
```

By adding the penetration depth to the moving object's position, it will move back outside the first object's collision box:

```
POSITION OBJECT 2,OBJECT POSITION X(2)+x#,
  ↳OBJECT POSITION Y(2)+y#,OBJECT POSITION Z(2)+z#
```

The program in LISTING-40.5 demonstrates the final effect.

LISTING-40.5

Using the GET OBJECT COLLISIONS Data to Modify the Path of an Object

```
REM *** Setup screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP 0
BACKDROP ON

REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 100,10,-20
POINT CAMERA 0,0,0

REM *** Make box ***
MAKE OBJECT BOX 1, 20,40,20

REM *** Create a bounding box taller than 3D box ***
MAKE OBJECT COLLISION BOX 1,-10,-20,-10,10,40,10,0

REM *** Make and position sphere ***
MAKE OBJECT SPHERE 2, 5
POSITION OBJECT 2,0,10,-20

REM *** Move sphere towards box until they collide ***
DO
  REM *** Move sphere 1 unit along z-axis ***
  POSITION OBJECT 2,OBJECT POSITION X(2),
  ↳OBJECT POSITION Y(2),OBJECT POSITION Z(2)+0.1
  REM *** IF sphere collides with box ***
  IF OBJECT COLLISION(1,2)
    REM *** Calculate penetration values ...***
    x#=GET OBJECT COLLISION X()
    y#=GET OBJECT COLLISION Y()
    z#=GET OBJECT COLLISION Z()
    REM *** ... and use them to reset sphere's position ***
    POSITION OBJECT 2,OBJECT POSITION X(2)+x#,
    ↳OBJECT POSITION Y(2)+y#,OBJECT POSITION Z(2)+z#
  ENDIF
LOOP
REM *** End program ***
END
```

Activity 40.15

Type in and test the coding in LISTING-40.5 (*collision05.dbpro*).

What happens when the sphere comes into contact with the cuboid?

The effect becomes more interesting when the sphere is moving with an offset in two or three axes. For example, we can make the sphere move in both the y and z axes by changing the first line within the DO loop to read:

```
POSITION OBJECT 2, OBJECT POSITION X(2) ,  
  ↵ OBJECT POSITION Y(2) -0.1, OBJECT POSITION Z(2) +0.1
```

Activity 40.16

Make the change suggested above to your last program. How does this affect the movement of the sphere?

This effect is known as a **sliding collision**, for obvious reasons, with the sphere appearing to slide along the surface of the cuboid until it's free to move along its old trajectory.

The DELETE OBJECT COLLISION BOX Statement

The collision box associated with a 3D object can be deleted using the DELETE OBJECT COLLISION BOX statement which has the format shown in FIG-40.22.

FIG-40.22

The DELETE OBJECT
COLLISION BOX
Statement



In the diagram:

objno

is an integer value specifying the object whose collision box is to be deleted.

Activity 40.17

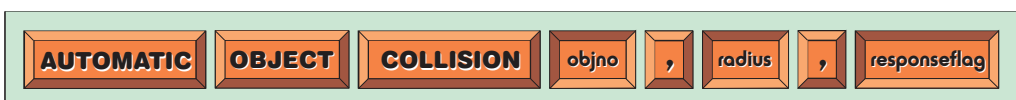
In your previous program, delete the cuboid's bounding box. Does the sphere halt because of collision?

The AUTOMATIC OBJECT COLLISION Statement

Another way of dealing with collisions is to use the AUTOMATIC OBJECT COLLISION statement. Using this statement allows an object to automatically stop moving when it collides with another object. When used, AUTOMATIC OBJECT COLLISION always uses a bounding sphere for the object specified. The radius of that sphere is set as part of this command. The statement has the format shown in FIG-40.23.

FIG-40.23

The AUTOMATIC OBJECT COLLISION Statement



In the diagram:

objno

is an integer value specifying the object whose collision sphere is to be specified.

radius

is a real number giving the radius of the collision sphere to be used by the object.

responseflag

is 0 or 1. If set to zero, the object halts as soon as a collision is detected; if set to 1, the object will move back to its previous position immediately before the collision.

The program in LISTING-40.6 demonstrates the use of this statement.

LISTING-40.6

Using Automatic Object Collision

```
REM *** Setup screen ***
SET DISPLAY MODE 1280, 1024, 32
COLOR BACKDROP 0
BACKDROP ON

REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 50,10,-5
POINT CAMERA 0,0,0

REM *** Make box ***
MAKE OBJECT BOX 1, 10,40,10

REM *** Make and position sphere ***
MAKE OBJECT SPHERE 2, 5
REM *** Use automatic collision detection ***
AUTOMATIC OBJECT COLLISION 2,2.5,0
POSITION OBJECT 2, 0,10,-100

REM *** Move sphere towards box until they collide ***
DO
  POSITION OBJECT 2,0,10,OBJECT POSITION Z(2)+1
  SET CURSOR 100,100
  PRINT OBJECT POSITION Z(2)
LOOP

REM *** End program ***
END
```

Activity 40.18

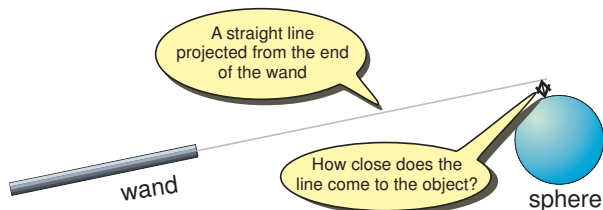
Type in and test the program given in LISTING-40.6 (*collision06.dbpro*).

The INTERSECT OBJECT Statement

Sometimes it is useful to know if one object will hit another object before the actual collision occurs. For example, will a bolt of lightning from a wand hit the *sphere of death* hurling towards our character (see FIG-40.24)?

FIG-40-24

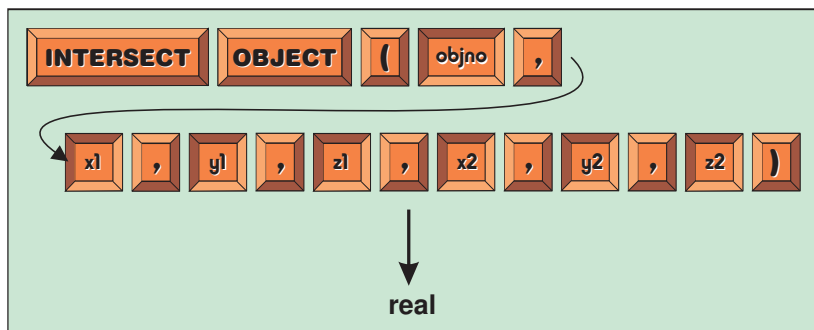
A Line Of Sight Test



To find out, we need some way of knowing that the direction in which the wand is pointing intersects with the sphere. The command that achieves this is INTERSECT OBJECT which has the format shown in FIG-40.25.

FIG-40.25

The INTERSECT
OBJECT Statement



In the diagram:

<i>objno</i>	is an integer value giving the target object.
<i>x1,y1,z1</i>	are a set of real values representing the starting point of the projected line.
<i>x2,y2,z2</i>	are a set of real values representing the finishing point of the projected line.

The statement returns the shortest distance between the target object and the line specified. If a value of zero is returned, the line and object intersect.

For the wand and sphere example mentioned earlier, *x1,y1,z1* would specify any point on or near the front of the wand. While *x2,y2,z2* would specify a point at the end of the range of the lightning bolt. The line created by joining these two points should be parallel to the body of the wand. *objno* would be the object number assigned to the sphere. This example is shown in LISTING-40.7.

LISTING-40.7

Intersection between an
Line and an Object

```

REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP 0
BACKDROP ON

REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 0,10,-100
POINT CAMERA 0,0,0

REM *** Make and position sphere ***
MAKE OBJECT SPHERE 1,10

REM *** Make and position wand ***
MAKE OBJECT CYLINDER 2,10
SCALE OBJECT 2, 10,200,10
POSITION OBJECT 2,-40,0,0
ZROTATE OBJECT 2,90

REM *** Projected line's start point is centre of wand ***
x1# = OBJECT POSITION X(2)
y1# = OBJECT POSITION Y(2)
z1# = OBJECT POSITION Z(2)
    
```

continued on next page

LISTING-40.7

(continued)

Intersection between an
Line and an Object

```
REM *** The wand fires a distance of 20 units ***
REM *** so end point of line is 20 units along x-axis ***
x2# = x1# + 20
y2# = y1#
z2# = z1#
dist# = INTERSECT OBJECT (2,x1#,y1#,z1#,x2#,y2#,z2#)

DO
    SET CURSOR 100,100
    PRINT dist#
LOOP

REM *** End program ***
END
```

Activity 40.19

Type in and test the program given in LISTING-40.7 (*collision07.dbpro*).

Summary

- Use OBJECT HIT or OBJECT COLLISION to determine if two objects have collided.
- Use SET OBJECT COLLISION to switch collision detection on or off for a specific object.
- Use SET GLOBAL COLLISION to switch collision detection for every object on or off.
- Collision detection uses a bounding box (also known as a collision box) or a bounding (collision) sphere.
- Collision detection using boxes and spheres is fast but can be inaccurate.
- The default setting is to use bounding boxes for collision detection.
- Collision with the actual polygons that make up the object itself can be used to give accurate collision detection, but requires much more complex calculations.
- Use SHOW OBJECT BOUNDS to show the bounding volumes of an object.
- Use HIDE OBJECT BOUNDS to hide the bounding volumes of an object.
- Use SET OBJECT COLLISION TO SPHERES to employ the bounding sphere of an object rather than its bounding box when detecting collisions.
- Use SET OBJECT RADIUS to modify the size of an object's bounding sphere.
- Use OBJECT COLLISION RADIUS to find out the current radius of an object's bounding sphere.
- Use SET OBJECT COLLISION TO BOXES to return to using the bounding box of an object to detect collisions.
- Use SET OBJECT COLLISION TO POLYGONS to employ an object's own polygons when detecting collisions.

- Use `MAKE OBJECT COLLISION BOX` to create your own bounding box for an object.
- Use the data from `GET OBJECT COLLISION` to create a sliding collision effect.
- Use `DELETE OBJECT COLLISION BOX` to delete any bounding box associated with a specified object.
- Use `AUTOMATIC OBJECT COLLISION` to have an object halt automatically when it strikes another object.
- Use `INTERSECT OBJECT` to determine if a line comes into contact with a specified object.

Static Collisions

Introduction

Some 3D objects in a game, such as buildings, walls, or trees will not move once they have been created. This type of object will be loaded from an existing file and become the scenery of a game. If moving objects are not to pass straight through this scenery, then that scenery needs to have specifically created bounding boxes. These collision boxes are known as **static bounding** (or **collision**) **boxes** since they will never move. DarkBASIC Pro has a set of commands for dealing with just such static collision boxes.

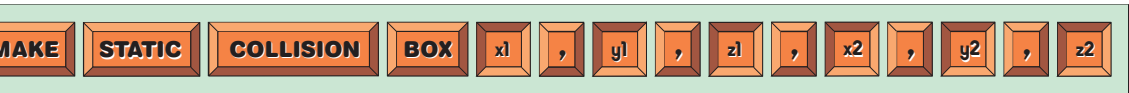
Creating and Using Static Collision Boxes

The MAKE STATIC COLLISION BOX Statement

We can create a static collision box using the MAKE STATIC COLLISION BOX statement whose format is given in FIG-40.26.

FIG-40.26

The MAKE STATIC COLLISION BOX Statement



In the diagram:

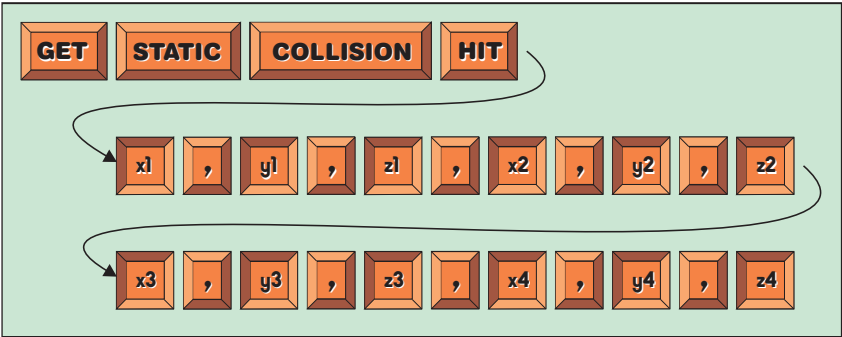
- $x1,y1,z1$
- are a set of real values giving the coordinates of the bottom-front, left corner of the collision box.
- $x2,y2,z2$
- are a set of values giving the coordinates of the top-back, right corner of the collision box.

The GET STATIC COLLISION HIT Statement

The usual OBJECT HIT and OBJECT COLLISION statements don't work with static collision boxes, so we need a new statement to detect a collision between an object and a static collision box. This statement is GET STATIC COLLISION HIT which has the format shown in FIG-40.27.

FIG-40.27

The GET STATIC COLLISION HIT Statement



In the diagram:

$x1,y1,z1$

area set of real values giving the coordinates of the bottom-front, left corner of the collision box.

$x2,y2,z2$

are a set of values giving the coordinates of the top-back, right corner of the collision box.

$x3,y3,z3$

are a set of real values giving the coordinates of the bottom-front, left corner of the collision box.

$x4,y4,z4$

are a set of values giving the coordinates of the top-back, right corner of the collision box.

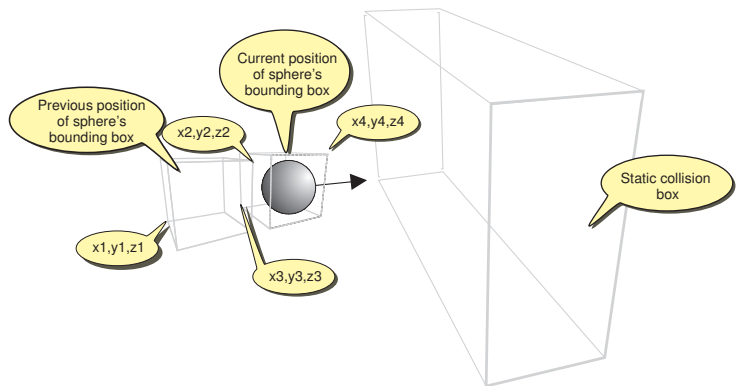
Notice that the statement makes no reference to an object - so how do we use it to detect when an object hits a static collision box?

Imagine a sphere travelling towards a static collision box as shown in FIG-40.28. To check if the sphere hits the box we need to specify two rectangular volumes; one for a position where the sphere had been a moment before, and one corresponding to the sphere's current position.

FIG-40.28

Using GET STATIC
COLLISION HIT

Notice that the trailing
bounding box is 3 units
behind the leading box.



The GET STATIC COLLISION HIT statement returns 1 if the leading box (that is, the box defined using $x3,y3,z3,x4,y4,z4$) intersects any part of the static collision box. If there is no intersection, zero is returned.

The program in LISTING-40.8 creates an invisible force-field, (that is, a static collision box). When a travelling sphere hits the force field, a string of asterisks is displayed on the screen. The line of asterisks grows with each hit detected. The ball is not deflected from its path.

LISTING-40.8

Hitting a Static Collision
Box

```
REM *** Setup screen ***
SET DISPLAY MODE 1280, 1024, 32
COLOR BACKDROP 0
BACKDROP ON

REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 50,10,-5
POINT CAMERA 0,0,0

REM *** Make static collision box ***
MAKE STATIC COLLISION BOX -20,0,-1,20,40,1
```

continued on next page

LISTING-40.8

(continued)

Hitting a Static Collision
Box

```
REM *** Make and position sphere ***
MAKE OBJECT SPHERE 2,5
POSITION OBJECT 2,0,10,-100

REM *** Record x and y ordinates of the old and new collision ***
REM *** boxes for the sphere. These don't change since the ***
REM *** sphere is moving in the z direction only ***
x1# = OBJECT POSITION X(2)-2.5
x2# = x1#+5
x3# = x1#
x4# = x2#
y1# = OBJECT POSITION Y(2)-2.5
y2# = y1#+5
y3# = y1#
y4# = y2#

REM *** Initialise string of asterisks ***
mess$ = ""

REM *** Main loop ***
DO
  REM *** Move sphere ***
  POSITION OBJECT 2,0,10,OBJECT POSITION Z(2)+1
  REM *** Get z ord of old & new bounding boxes for sphere ***
  z1# = OBJECT POSITION Z(2)-3.5
  z2# = z1# + 5
  z3# = z1# + 1
  z4# = z2# + 1
  REM *** Check bounding box hit with static collision box ***
  IF GET STATIC COLLISION HIT(x1#,y1#,z1#,x2#,y2#,z2#,x3#,y3#,z3#
  ,x4#,y4#,z4#)
    REM *** Add to number of asterisks ***
    mess$=mess$ + "*"
  ENDIF
  REM *** Display message ***
  SET CURSOR 100,100
  PRINT mess$
LOOP

REM *** End program ***
END
```

Activity 40.20

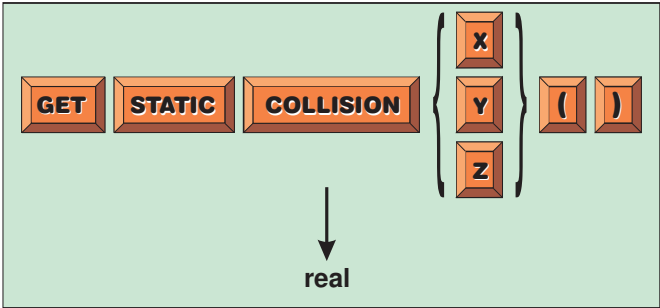
Type in and test the program in LISTING-40.8 (*collision08.dbpro*).

The GET STATIC COLLISION Statement

When the GET STATIC COLLISION HIT statement is used, the extent to which the specified bounding cube has penetrated the static collision box is recorded. This information can be retrieved using the GET STATIC COLLISION statement which has the format given in FIG-40.29.

FIG-40.29

The GET STATIC
COLLISION Statement



In the diagram:

X,Y,Z

Use one of these options to determine the penetration of the bounding box within the static collision box in the specified direction.

Normally, we would use all three options to get the full details of the collision:

```
x# = GET STATIC COLLISION X()
y# = GET STATIC COLLISION Y()
z# = GET STATIC COLLISION Z()
```

The next program (LISTING-40.9) is a modification of the last one. This time the asterisks are replaced by the values returned by GET STATIC COLLISION statements. Also, the display uses the SYNC statement so that we can pause the program with a WAIT KEY statement and still see the values output by the PRINT statement.

LISTING-40.9

Understanding the Value
Returned by GET
STATIC COLLISION

```
REM *** Setup screen ***
SET DISPLAY MODE 1280, 1024, 32
COLOR BACKDROP 0
BACKDROP ON
REM *** Use manual screen updating ***
SYNC ON

REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 50,10,-5
POINT CAMERA 0,0,0

REM *** Make static collision box ***
MAKE STATIC COLLISION BOX -20,0,-1,20,40,1

REM *** Make and position sphere ***
MAKE OBJECT SPHERE 2, 5
POSITION OBJECT 2,0,10,-100

REM *** Record x and y of old and new bounding boxes ***
x1# = OBJECT POSITION X(2)-2.5
x2# = x1#+5
x3# = x1#
x4# = x2#
y1# = OBJECT POSITION Y(2)-2.5
y2# = y1#+5
y3# = y1#
y4# = y2#

REM *** Main loop ***
DO
  REM *** Move sphere ***
  POSITION OBJECT 2,0,10,OBJECT POSITION Z(2)+1
  REM *** Calculate z ordinate for old and new boxes ***
  z1# = OBJECT POSITION Z(2)-5.5
  z2# = z1# + 5
  z3# = z1# + 3
  z4# = z2# + 3
  REM *** Check for hit with static collision box ***
  IF GET STATIC COLLISION HIT(x1#,y1#,z1#,x2#,y2#,z2#,x3#
  ,y3#,z3#,x4#,y4#,z4#)
    REM *** Get collision details ***
    x# = GET STATIC COLLISION X()
    y# = GET STATIC COLLISION Y()
    z# = GET STATIC COLLISION Z()
```

continued on next page

LISTING-40.9

(continued)

Understanding the Value
Returned by GET
STATIC COLLISION

```
REM *** Display these details ***
SET CURSOR 100,100
PRINT x#," ",y#," ",z#,"          ",z1#," ",z2#,"          ",z3#,"
  ",z4#
REM *** Update screen ***
SYNC
REM *** Wait for key press ***
WAIT KEY
ENDIF
REM *** Update screen ***
SYNC
LOOP

REM *** End program ***
END
```

Activity 40.21

Type in and test the program in LISTING-40.9 (*collision09.dbpro*), observing the figures displayed.

As we can see from the figures, as the leading bounding box enters the collision area the value returned by GET STATIC COLLISION Z is determined by the result of the calculation

*leading bounding box's z ordinate (from the leading edge) -
static collision box's z ordinate on the entry side*

which is simply:

$$z4 - 1$$

Activity 40.22

Change the first z value of the static collision box from -1 to -2 (thereby making the static collision box 3 units thick) and run the program again.

Notice that the last value displayed by GET STATIC COLLISION Z is -0.5. Once the mid-point of the trailing bounding box is passed the mid-point of the static collision box, the formula for the value returned by GET STATIC COLLISION Z changes to:

*static collision box's z ordinate on the entry side -
leading bounding box's trailing edge z-ordinate*

In this case that is

$$-2 - z3$$

Activity 40.23

Check that the formula still holds by

- Changing the static collision box's z ordinates from -2 and 1 to -5 and 10.
- Making the sphere enter from the other side of the static collision box (that is, start the sphere at position (0,10,100) and give it a speed of -1).

Similarly, if the sphere moves parallel to the x-axis, then it is the GET STATIC COLLISION X statement which will return a non-zero value based on the bounding box's leading edge's x value. Approaching parallel to the y-axis, GET STATIC COLLISION Y returns a non-zero value.

So how can we make use of the value returned by the GET STATIC COLLISION values in our program? Well, if we use the value returned by the statement to modify the position of our moving object, we can stop that object moving through the static collision box area.

Activity 40.24

Modify the DO loop in your previous program to read

```
DO
  REM *** Move sphere ***
  POSITION OBJECT 2,0,10,OBJECT POSITION Z(2)+1
  REM *** Calculate z ordinate for old and new boxes ***
  z1# = OBJECT POSITION Z(2)-5.5
  z2# = z1# + 5
  z3# = z1# + 3
  z4# = z2# + 3
  REM *** Check for collision with static collision box ***
  IF GET STATIC COLLISION HIT (x1#,y1#,z1#,x2#,y2#,z2#,
  Åx3#,y3#,z3#,x4#,y4#,z4#)
    REM *** Get penetration details ***
    x# = GET STATIC COLLISION X()
    y# = GET STATIC COLLISION Y()
    z# = GET STATIC COLLISION Z()
    REM *** Move sphere back out of static collision box ***
    POSITION OBJECT 2,0,10,OBJECT POSITION Z(2) - z#
  ENDIF
  REM *** Update screen ***
  SYNC
LOOP
```

Position the sphere at location (0,10,-100).

Run the program. What happens when the sphere hits the static box?

As you have seen, we can use the GET STATIC COLLISION value to force the sphere to stop when it hits the static collision box.

When the moving object contains movement along a second axis, then the overall effect is to slide the moving object along the surface of the static collision box.

We see an example of this in LISTING-40.10 where the sphere's movement has both a y and z component. As the sphere hits the static collision box, it slides down the side of that box until it passes underneath and then continues on its way.

LISTING-40.10

Using the Value
Returned by GET
STATIC COLLISION

```
REM *** Setup screen ***
SET DISPLAY MODE 1280, 1024, 32
COLOR BACKDROP 0
BACKDROP ON
REM *** Use manual screen updating ***
SYNC ON
```

continued on next page

LISTING-40.10

(continued)

Using the Value
Returned by GET
STATIC COLLISION

```
REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 50,10,-15
POINT CAMERA 0,0,0
REM *** Make static collision box ***
MAKE STATIC COLLISION BOX -20,0,-5,20,40,10

REM *** Make and position sphere ***
MAKE OBJECT SPHERE 2, 5
POSITION OBJECT 2,0,10,-100

REM *** Record x for old and new bounding boxes for sphere ***
x1# = OBJECT POSITION X(2)-2.5
x2# = x1#+5
x3# = x1#
x4# = x2#

REM *** Main loop ***
DO
  REM *** Move sphere ***
  POSITION OBJECT 2,0,OBJECT POSITION Y(2)-0.05,
  ↳ OBJECT POSITION Z(2)+1
  REM *** Calculate y and z ordinate for old and new boxes ***
  y1# = OBJECT POSITION Y(2)-2.5
  y2# = y1#+5
  y3# = y1#
  y4# = y2#
  z1# = OBJECT POSITION Z(2)-5.5
  z2# = z1# + 5
  z3# = z1# + 3
  z4# = z2# + 3
  REM *** Check collision with static collision box ***
  IF GET STATIC COLLISION HIT(x1#,y1#,z1#,x2#,y2#,z2#,x3#,y3#,z3#
  ↳ ,x4#,y4#,z4#)=1
    REM *** Get penetration details ***
    x# = GET STATIC COLLISION X()
    y# = GET STATIC COLLISION Y()
    z# = GET STATIC COLLISION Z()
    REM *** Move sphere back out of static collision box ***
    POSITION OBJECT 2,0,OBJECT POSITION Y(2) - y#,
    ↳ OBJECT POSITION Z(2) - z#
    REM *** Display message ***
  ENDIF
  SYNC
LOOP
REM *** End program ***
END
```

Activity 40.25

Type in and test the program given in LISTING-40.10 (*collision10.dbpro*).

As a general rule, since we may not know through which axes an object is moving, the values *x1#* to *z4#* should be calculated on every iteration of the main loop and the repositioning after a hit should include all three penetration figures:

```
POSITION OBJECT 2,OBJECT POSITION X()-x#,
↳ OBJECT POSITION Y(2)-y#,OBJECT POSITION Z(2)-z#
```

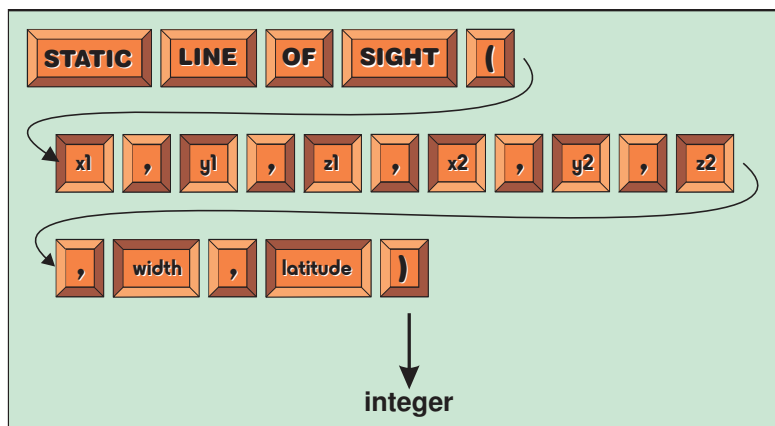
The STATIC LINE OF SIGHT Statement

When we look down the barrel of a gun, we will want to know if we are going to hit our intended target before we actually fire. DarkBASIC Pro can answer this type

of question, determining if a given trajectory will intersect with a static collision box, using the STATIC LINE OF SIGHT statement which has the format shown in FIG-40.30.

FIG-40.30

The STATIC LINE OF SIGHT Statement



In the diagram:

- | | |
|-----------------|--|
| $x1, y1, z1$ | are a set of real values giving the start point of a line. |
| $x2, y2, z2$ | are a set of real values giving the end point of a line. |
| <i>width</i> | is a real number giving the width of the line. |
| <i>latitude</i> | is a real number affecting the degree of accuracy of the calculation performed by this statement. A large number represents greater degree of latitude in the calculation. |

This statement returns 1 if the line starting at point ($x1, y1, z1$) and passing through the point ($x2, y2, z2$) eventually hits an existing static collision box; otherwise zero is returned. In performing the calculation the line is assumed to be in the form of a cylinder with diameter *width*.

Assuming a weapon's centre is at point (-40,0,0) and that the weapon has been rotated by 90° about the z-axis, with one point along the line of sight being (-5,0,0), then we can check if this line of sight hits a static collision box using the line:

```
IF STATIC LINE OF SIGHT (-40,0,0,-5,0,0.5,1,1) = 1
```

In this example, the line has been given a thickness of 1 and the latitude setting is also set to 1 (thereby specifying a high degree of accuracy in the calculation).

The program in LISTING-40.11 demonstrates the use of this statement.

LISTING-40.11

Using the STATIC LINE OF SIGHT Statement

```

REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP 0
BACKDROP ON

REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 0,10,-100
POINT CAMERA 0,0,0
  
```

continued on next page

LISTING-40.11

(continued)

Using the STATIC LINE
OF SIGHT Statement

```
REM *** Make static collision box and visible box ***
MAKE STATIC COLLISION BOX -20,-20,-20,20,20,20
MAKE OBJECT BOX 1,40,40,40

REM *** Make and position wand ***
MAKE OBJECT CYLINDER 2,10
SCALE OBJECT 2, 10,200,10
POSITION OBJECT 2,-40,0,0
ZROTATE OBJECT 2,90

REM *** Discover if wand is pointing at static box ***
REM *** using two points along line of wand ***
ans = STATIC LINE OF SIGHT (-40,0,0,-5,0,0,0,1)

REM *** Display result ***
DO
    set cursor 100,100
    PRINT ans
LOOP

REM *** End program ***
END
```

Activity 40.26

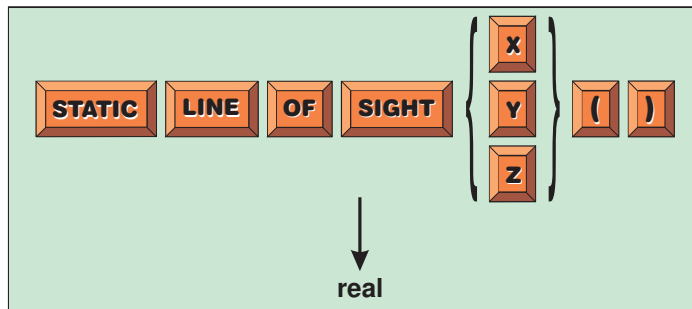
Type in and test the program given in LISTING-40.11 (*collision11.dbpro*).

The STATIC LINE OF SIGHT Coordinates Statement

We can discover the approximate coordinates of the point of contact between a line of sight and a static collision box using the STATIC LINE OF SIGHT coordinates statement which has the format shown in FIG-40.31.

FIG-40.31

The STATIC LINE OF
SIGHT Coordinates
Statement



In the diagram:

X,Y,Z

Use one of these options to determine the ordinate of the point of contact with the static collision box for the required axis.

For example, we could record the point of contact between the line of sight and a static collision box with the lines:

```
IF STATIC LINE OF SIGHT (-40,0,0,-5,-0.2,0.5,0,1)= 1
  x# = STATIC LINE OF SIGHT X()
  y# = STATIC LINE OF SIGHT Y()
  z# = STATIC LINE OF SIGHT Z()
```

Activity 40.27

Modify your previous program to display the point of intersection between the light of sight and static collision box.

Static Collision Boxes and the Camera

Back in Chapter 33 we had a look at the SET CAMERA TO FOLLOW statement. The last parameter to this statement is a flag which determines how the camera reacts to collision with a static collision box. Set to 0, the camera will ignore any static collision boxes; set to 1, the camera should slide past such boxes.

The program in LISTING-40.12 creates a camera and a static collision box. The camera is then moved to a new position using the SET CAMERA TO FOLLOW statement and this requires the camera to come into contact with the collision box.

LISTING-40.12

The SET CAMERA TO FOLLOW Statement and Static Collision Boxes

```
REM *** Setup screen ***
SET DISPLAY MODE 1280, 1024, 32
COLOR BACKDROP RGB(255,255,100)
BACKDROP ON

REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 0,0,-50
POINT CAMERA 0,0,0

REM *** Make static collision box ***
MAKE STATIC COLLISION BOX -20,-20,-3,20,20,3

REM *** Make visible object over collision box ***
MAKE OBJECT BOX 1,40,40,6

REM *** Try to move camera through collision box ***
DO
    SET CAMERA TO FOLLOW 0,20,15,5,2,2,100,0
    WAIT 50
LOOP

REM *** End program ***
END
```

Activity 40.28

Type in and run the program given above (*collision12.dbpro*).

Does the camera pass through the box?

Change the final value of the SET CAMERA TO FOLLOW statement to 1 and re-run the program.

How does the camera react this time?

Summary

- Static bounding (collision) boxes are used to create a detectable collision volume around fixed areas.
- Use MAKE STATIC COLLISION BOX to create a static collision box.

- Use GET STATIC COLLISION HIT to detect collision with a static bounding box.
- Use GET STATIC COLLISION to determine the penetration of an object into a static collision box.
- Data from GET STATIC COLLISION can be used to create a sliding collision of an object against a static bounding box.
- Use STATIC LINE OF SIGHT to determine if an imaginary line will pass through a static bounding box.
- Use STATIC LINE OF SIGHT coordinates to determine a line of sight's contact point with a static bounding box.
- The SET CAMERA TO FOLLOW statement should automatically slide over any static collision box when the final parameter is set to 1.

Solutions

Activity 40.1

The sphere passes straight through the cube.

```
CONTROL CAMERA USING ARROWKEYS 0, 0.1,
↳ 0.5
LOOP
REM *** End program ***
END
```

Activity 40.2

```
REM *** Setup screen ***
SET DISPLAY MODE 1280, 1024, 32
COLOR BACKDROP 0
BACKDROP ON
REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 100,10,-200
POINT CAMERA 0,0,0
REM *** Make two objects ***
MAKE OBJECT CUBE 1, 40
MAKE OBJECT SPHERE 2, 5
POSITION OBJECT 2, 0,0,-100
REM *** Move sphere towards cube ***
DO
    POSITION OBJECT 2,0,0,
    OBJECT POSITION Z(2)+1
    REM *** Check for collision ***
    IF OBJECT HIT(1,2)
        EXIT
    ENDIF
LOOP
REM *** Give user camera control ***
DO
    CONTROL CAMERA USING ARROWKEYS 0, 0.1,0.5
LOOP
REM *** End program ***
END
```

Activity 40.3

The first DO..LOOP structure should be changed to read:

```
DO
    POSITION OBJECT 2,0,0,
    ↳ OBJECT POSITION Z(2)+0.1
    IF OBJECT COLLISION(1,2)
        EXIT
    ENDIF
LOOP
```

Activity 40.4

```
REM *** Setup screen ***
SET DISPLAY MODE 1280, 1024, 32
COLOR BACKDROP 0
BACKDROP ON
REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 100,10,-200
POINT CAMERA 0,0,0
REM *** Make two objects ***
MAKE OBJECT CUBE 1, 40
MAKE OBJECT SPHERE 2, 5
POSITION OBJECT 2, 0,0,-100
REM *** Switch off cube's col'sn detect ***
SET OBJECT COLLISION OFF 1
REM *** Move sphere towards cube ***
DO
    POSITION OBJECT 2,0,0,
    ↳ OBJECT POSITION Z(2)+1
    IF OBJECT COLLISION(1,2)
        EXIT
    ENDIF
LOOP
DO
```

Activity 40.5

```
REM *** Setup screen ***
SET DISPLAY MODE 1280, 1024, 32
COLOR BACKDROP 0
BACKDROP ON
REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 100,10,-200
POINT CAMERA 0,0,0
REM *** Make two objects ***
MAKE OBJECT CUBE 1, 40
MAKE OBJECT SPHERE 2, 5
POSITION OBJECT 2, 0,0,-100
REM *** Switch off cube's col'sn detect ***
SET OBJECT COLLISION OFF 1
REM *** Switch on global collisions ***
SET GLOBAL COLLISION ON
REM *** Move sphere towards cube ***
DO
    POSITION OBJECT 2,0,0,
    ↳ OBJECT POSITION Z(2)+1
    IF OBJECT COLLISION(1,2)
        EXIT
    ENDIF
LOOP
DO
    CONTROL CAMERA USING ARROWKEYS 0, 0.1,
    ↳ 0.5
LOOP
REM *** End program ***
END
```

Activity 40.6

```
REM *** Setup screen ***
SET DISPLAY MODE 1280, 1024, 32
COLOR BACKDROP 0
BACKDROP ON
REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 100,10,-200
POINT CAMERA 0,0,0
REM *** Make two objects ***
MAKE OBJECT CUBE 1, 40
MAKE OBJECT SPHERE 2, 5
POSITION OBJECT 2, 0,0,-100
REM *** Show bounding boxes ***
SHOW OBJECT BOUNDS 1,1
SHOW OBJECT BOUNDS 2,1
REM *** Move sphere towards cube ***
DO
    POSITION OBJECT 2,0,0,
    ↳ OBJECT POSITION Z(2)+1
    IF OBJECT COLLISION(1,2)
        EXIT
    ENDIF
LOOP
DO
    CONTROL CAMERA USING ARROWKEYS 0, 0.1,
    ↳ 0.5
LOOP
REM *** End program ***
END
```

Activity 40.7

The objects should now be created using the lines:

```
REM *** Make two objects ***
MAKE OBJECT CONE 1, 40
MAKE OBJECT SPHERE 2,5
```

A collision is detected long before the sphere actually collides with the cone.

Activity 40.8

To hide the bounds the following changes could be made:

```
REM *** Display bounding volumes ***
SHOW OBJECT BOUNDS 1,1
SHOW OBJECT BOUNDS 2,1
REM *** Hide bounding boxes ***
WAIT KEY
HIDE OBJECT BOUNDS 1
HIDE OBJECT BOUNDS 2
```

Activity 40.9

No solution required.

Activity 40.10

```
REM *** Setup screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP 0
BACKDROP ON
REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 100,10,-200
POINT CAMERA 0,0,0
REM *** Make box ***
MAKE OBJECT BOX 1,10,60,10
REM *** Display its bounding volumes ***
SHOW OBJECT BOUNDS 1
REM *** Lie box on its side ***
ZROTATE OBJECT 1, 90
REM *** Create a sphere ***
MAKE OBJECT SPHERE 2, 5
SHOW OBJECT BOUNDS 2
REM *** Position sphere ***
POSITION OBJECT 2,0,40,-100
REM *** Use Box's bounding sphere ***
SET OBJECT COLLISION TO SPHERES 1
REM *** Change bounding sphere radius ***
SET OBJECT RADIUS 1,50
REM *** Move sphere towards box ***
DO
    POSITION OBJECT 2,0,20,
    ↵OBJECT POSITION Z(2)+1
    IF OBJECT HIT(1,2)
        EXIT
    ENDIF
LOOP
REM *** End program ***
WAIT KEY
END
```

The 3D sphere stops when it collides with the cuboid's bounding sphere.

Activity 40.11

Two lines need to be changed to lift the sphere to 40 units. Change

```
POSITION OBJECT 2,0,20,-100
```

to

```
POSITION OBJECT 2,0,40,-100
```

Change

```
POSITION OBJECT 2,0,20,
↵OBJECT POSITION Z(2)+1
```

to

```
POSITION OBJECT 2,0,40,
↵OBJECT POSITION Z(2)+1
```

The cube flies past the box without colliding.

The version of the program needed to increase the bounding sphere is given below:

```
REM *** Setup screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP 0
BACKDROP ON
REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 100,10,-200
POINT CAMERA 0,0,0
REM *** Make box ***
MAKE OBJECT BOX 1,10,60,10
REM *** Display its bounding volumes ***
SHOW OBJECT BOUNDS 1
REM *** Lie box on its side ***
ZROTATE OBJECT 1, 90
REM *** Create a sphere ***
MAKE OBJECT SPHERE 2, 5
SHOW OBJECT BOUNDS 2
REM *** Position sphere ***
POSITION OBJECT 2,0,40,-100
REM *** Use Box's bounding sphere ***
SET OBJECT COLLISION TO SPHERES 1
REM *** Change bounding sphere radius ***
SET OBJECT RADIUS 1,50
REM *** Move sphere towards box ***
DO
    POSITION OBJECT 2,0,40,
    ↵OBJECT POSITION Z(2)+1
    IF OBJECT HIT(1,2)
        EXIT
    ENDIF
LOOP
REM *** End program ***
WAIT KEY
END
```

Activity 40.12

Neither method gives accurate collision detection.

Activity 40.13

The sphere correctly collides with the cone.

Activity 40.14

The sphere stops as it hits the extended collision box above the cuboid.

Rotating the cuboid does not rotate the collision box and therefore a collision still occurs.

By changing the last value in the MAKE OBJECT

COLLISION BOX to 1, the collision box rotates along with the cuboid and so the sphere passes without a collision occurring.

Activity 40.15

The sphere halts when it collides with the cuboid.

Activity 40.16

The sphere slides down the side of the cuboid until it reaches the bottom and then continues on its path.

Activity 40.17

Without the bounding box, the sphere should travel straight through the cuboid but it appears that DELETE OBJECT COLLISION BOX does not operate correctly and the collision box remains (DBPro v 1.062).

Activity 40.18

No solution required.

Activity 40.19

No solution required.

Activity 40.20

No solution required.

Activity 40.21

No solution required.

Activity 40.22

No solution required.

Activity 40.23

The formula still holds in both cases.

Activity 40.24

The sphere halts when the static collision box is encountered.

Activity 40.25

No solution required.

Activity 40.26

No solution required.

Activity 40.27

```
REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP 0
BACKDROP ON
```

```
REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 0,10,-100
POINT CAMERA 0,0,0
REM *** Make static collision box
REM *** and visible box ***
MAKE STATIC COLLISION BOX
↳-20,-20,-20,20,20,20
MAKE OBJECT BOX 1,40,40,40
REM *** Make and position wand ***
MAKE OBJECT CYLINDER 2,10
SCALE OBJECT 2, 10,200,10
POSITION OBJECT 2,-40,0,0
ZROTATE OBJECT 2,90
REM *** If line of sight collision ***
REM *** get coordinates ***
IF STATIC LINE OF SIGHT
↳(-40,0,0,-5,0,0,0,1) = 1
  x# = STATIC LINE OF SIGHT X()
  y# = STATIC LINE OF SIGHT Y()
  z# = STATIC LINE OF SIGHT Z()
ENDIF
REM *** Display result ***
DO
  SET CURSOR 100,100
  PRINT "Point of intersection : ",x#," "
  ↳,y#," ",z#
LOOP
REM *** End program ***
END
```

Activity 40.28

The camera passes straight through the box (and collision box).

The camera STILL seems to pass through the collision box!

Fire Particles

Fountain Particles

Snow Particles

Setting Particle Characteristics

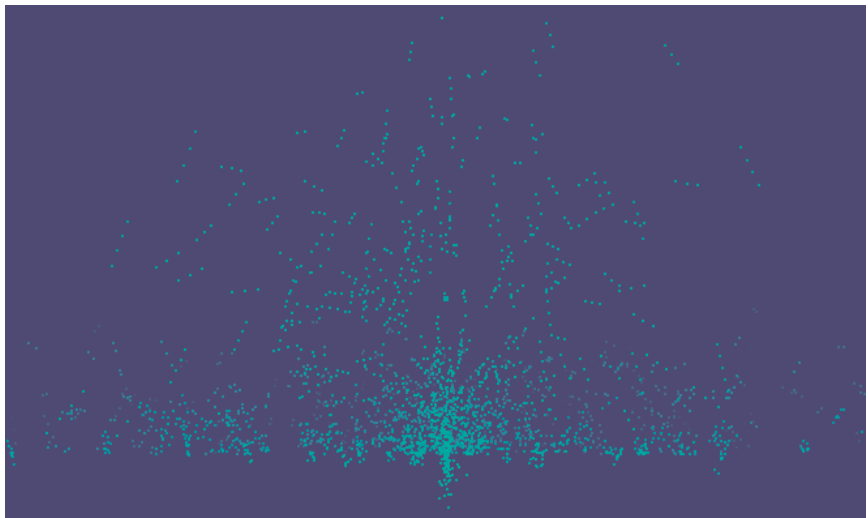
Using Particles

Introduction

If you've ever seen a sparkler in action on Guy Fawkes night (or whatever day your country uses as an excuse to let us play with fireworks) you'll be able to imagine the effect of the particle instructions in DarkBASIC Pro. Once you've created a particles object it continues to create a sparkler effect until it is destroyed. A snapshot of the basic effect is shown in FIG-41.1.

FIG-41.1

Particles in Action



The particles shoot upwards and then fall back to an imaginary ground area before eventually fading.

There are various characteristics that can be set, such as the quantity, speed, and duration of the particles. It will be useful if we think of a particle object as having two components: the emitter, and the particles themselves. The particles all originate from the emitter, which is at the centre of the particle display.

Creating Particles

The MAKE PARTICLES Statement

To create a stream of particles we'll need an image. This image is used on each particle created, but seems to have only a minor visual effect on what you'll see on the screen.

With an image loaded, you can now use the MAKE PARTICLES statement to create a continual stream of particles which originate from a single point (called the **emitter**), fly upwards and then fall back down before finally disappearing. The statement has the format shown in FIG-41.2.

FIG-41.2

The MAKE PARTICLES Statement



In the diagram:

partno

is an integer value specifying the ID assigned to this particle object.

imgno

is an integer value previously assigned to a loaded image. The image has a minor effect on the appearance of the particles.

partflag

is an integer value which affects the quantity of particles produced. Use zero to produce no particles; any other positive integer produces particles.

size

is a real number specifying how far out the particles will fly from their origin and also affects the size of the particles.

The particles object's emitter is automatically positioned at (0,0,0), but may be moved after it has been created.

The program in LISTING-41.1 demonstrates the use of the MAKE PARTICLE statement.

LISTING-41.1

Creating a Particles
Object

```
REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP 0
BACKDROP ON

REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 0,0,5,-10

REM *** Load image for particles ***
LOAD IMAGE "green.bmp",1

REM *** Create a particles object ***
MAKE PARTICLES 1,1,1,1

REM *** End program ***
WAIT KEY
END
```

Activity 41.1

Type in and test the program given above (*particles01.dbpro*).

Change the value of *size* to 10 and see how this affects the particles.

The HIDE PARTICLES Statement

If we want to hide existing particles, then we can use the HIDE PARTICLES statement which has the format shown in FIG-41.3.

FIG-41.3

The HIDE PARTICLES
Statement



In the diagram:

partno

is the integer value specifying the ID of the particles object to be hidden.

Activity 41.2

Modify *particles01.dbpro* so that the particle object disappears when a key is pressed.

The SHOW PARTICLES Statement

If you want the particles object to reappear after being hidden, then use the SHOW PARTICLES statement which has the format shown in FIG-41.4.

FIG-41.4

The SHOW
PARTICLES Statement



In the diagram:

partno

is the integer value identifying the previously hidden particles object which is to reappear.

The DELETE PARTICLES Statement

When you've no more need for a particles object, use the DELETE PARTICLES statement which will free up the RAM space reserved for the object. The statement has the format shown in FIG-41.5.

FIG-41.5

The DELETE
PARTICLES Statement



In the diagram:

partno

is the integer value identifying the particles object which is to be deleted.

The POSITION PARTICLES Statement

Like any other 3D object, the particles object's emitter is initially placed at location (0,0,0). To move the particles object to a different position in 3D space, use the POSITION PARTICLES statement which takes the format shown in FIG-41.6.

FIG-41.6

The POSITION
PARTICLES Statement



In the diagram:

partno

is the integer value identifying the particles object which is to be repositioned.

x,y,z

are the coordinates at which the particles object's emitter is to be placed.

This statement moves the whole particles object's emitter and currently falling particles to a new location. The program in LISTING-41.2 demonstrates this effect.

LISTING-41.2

Repositioning the
Particles

```
REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP 0
BACKDROP ON

REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 0,0,0,-16

REM *** Load image for particles ***
LOAD IMAGE "yellow.bmp",1

REM *** Create particles object ***
MAKE PARTICLES 1,1,1,1

REM *** Move position of particles object several times ***
post# = -10
FOR c = 1 TO 200
    POSITION PARTICLES 1,post#,0,0
    post# = post# + 0.1
    WAIT 10
NEXT c

REM *** End program ***
WAIT KEY
END
```

Activity 41.3

Type in and test the program given above (*particles02.dbpro*).

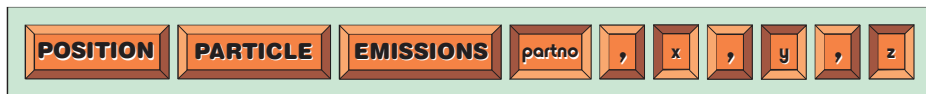
Modify the code so that the particles object is moved up the screen rather than from left to right.

The POSITION PARTICLE EMISSIONS Statement

Whereas the POSITION PARTICLES statement moves the complete particles object, the POSITION PARTICLE EMISSIONS statement moves only the emitter point and leaves any existing particles in their original position, still falling. New particles are also generated from the new emitter position. This statement has the format shown in FIG-41.7.

FIG-41.7

The POSITION
PARTICLE
EMISSIONS Statement



In the diagram:

partno

is the integer value previously assigned to the particles object.

x,y,z

are the coordinates at which the particles object's emitter is to be placed.

The program in LISTING-41.3 only differs from the previous program by a single line of code, but the effect on the screen is quite different.

LISTING-41.3

Moving the Particles
Emitter

```
REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP 0
BACKDROP ON

REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 0,0,0,-16

REM *** Load image for particles ***
LOAD IMAGE "yellow.bmp",1

REM *** Create particles object ***
MAKE PARTICLES 1,1,1,10

REM *** Move position of particles object several times ***
post#= -10
FOR c = 1 TO 200
    POSITION PARTICLE EMISSIONS 1,post#,0,0
    post# = post# + 0.1
    WAIT 10
NEXT c

REM *** End program ***
WAIT KEY
END
```

Activity 41.4

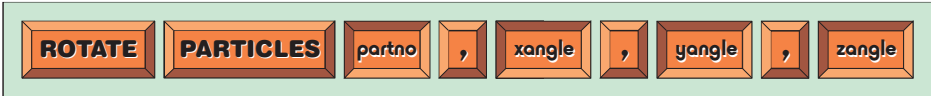
Modify your previous program to match the code given above and check out the difference between using POSITION PARTICLES and POSITION PARTICLE EMISSIONS.

The ROTATE PARTICLES Statement

The emitter can be rotated about its position using the ROTATE PARTICLES statement. This has the effect of changing the direction in which the particles are fired and the plane in which they bounce. The format of the ROTATE PARTICLE statement is shown in FIG-41.8.

FIG-41.8

The ROTATE
PARTICLES
Statement

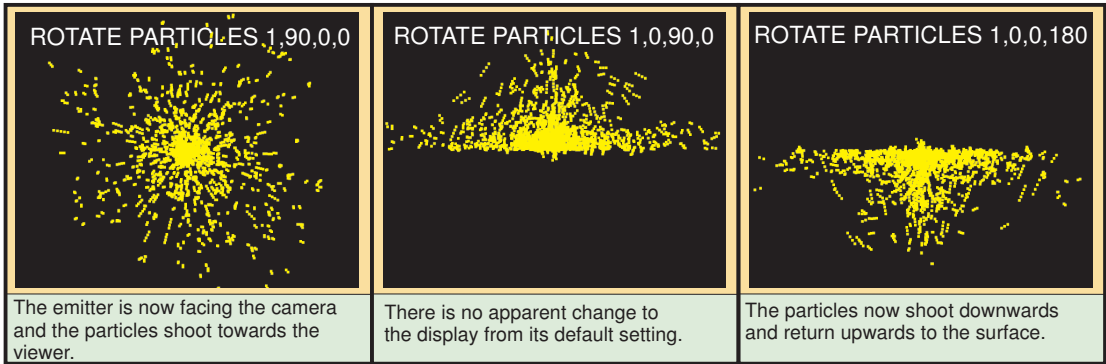


In the diagram:

- | | |
|---------------|---|
| <i>partno</i> | is the integer value previously assigned to the particles object. |
| <i>xangle</i> | is a real value giving the angle (in degrees) to which the emitter is rotated about the x-axis. |
| <i>yangle</i> | is a real value giving the angle (in degrees) to which the emitter is rotated about the y-axis. |
| <i>zangle</i> | is a real value giving the angle (in degrees) to which the emitter is rotated about the z-axis. |

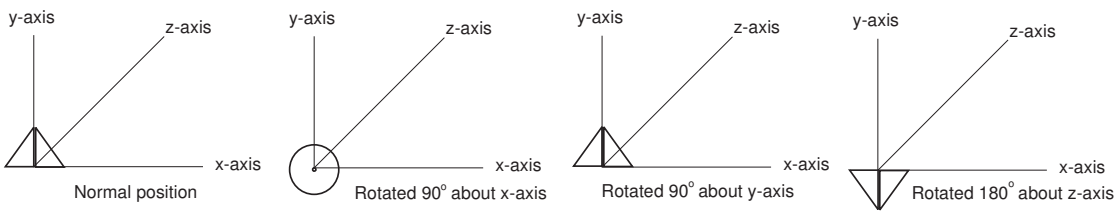
The effects of various ROTATE PARTICLES statements are shown in FIG-41.9.

FIG-41.9 The Effects of Rotating the Particle Emitter



It's easier to get your head round this if you think of the whole particles effect as creating a cone-shaped display, with the particles rising straight up through the centre of the cone and then falling back round its base perimeter. Now we can show the effects of each rotation in a more abstract way (see FIG-41.10).

FIG-41.10 How the Emitter is Rotated



The COLOR PARTICLES Statement

You can modify the colour of the particles emitted using the COLOR PARTICLES statement which has the format shown in FIG-41.11.

FIG-41.11

the COLOR PARTICLES Statement



In the diagram:

<i>partno</i>	is the integer value previously assigned to the particles object.
<i>red</i>	is an integer value specifying the intensity of the red component in the particles.
<i>green</i>	is an integer value specifying the intensity of the green component in the particles.
<i>blue</i>	is an integer value specifying the intensity of the blue component in the particles

The particles are actually set to a colour determined by the colour in the original image plus the colour specified here. By using a pure white image when setting up the particles object, the COLOR PARTICLES statement causes the particles to

match the colour stated exactly.

To set the particles to yellow we could use the statement:

```
COLOR PARTICLES 1, 255,255,0
```

The program in LISTING-41.4 randomly changes the colour of the particles every second for 10 seconds.

LISTING-41.4

Changing Particle Colour

```
REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP 0
BACKDROP ON

REM *** Set up camera ***
AUTOCAM OFF
POSITION CAMERA 0, 0,0,-10

REM *** Set up particle object ***
LOAD IMAGE "white.bmp",1
MAKE PARTICLES 1,1,1,10

REM *** Seed randomizer ***
RANDOMIZE TIMER()

REM *** Change colour 10 times ***
FOR c = 1 TO 10
    COLOR PARTICLES 1, RND(255),RND(255),RND(255)
    WAIT 1000
NEXT c
REM *** End program ***
WAIT KEY
END
```

Activity 41.5

Type in and test the program given above (*particles03.dbpro*).

The SET PARTICLE EMISSIONS Statement

The quantity of particles produced by the emitter can be modified using the SET PARTICLE EMISSIONS statement which has the format shown in FIG-41.12.

FIG-41.12

The SET PARTICLE EMISSIONS Statement



In the diagram:

partno

is the integer value previously assigned to the particles object.

value

is an integer value which modifies the quantity of particles produced by the emitter.

This statement has no effect on the height or distance over which the particles travel.

A typical statement might be:

```
SET PARTICLE EMISSIONS 1,12
```

The program in LISTING-41.5 demonstrates the effect of this statement by gradually increasing the particle emissions at one second intervals.

LISTING-41.5

The Effects of Particle Emissions

```
REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP 0
BACKDROP ON

REM *** Set up camera ***
AUTOCAM OFF
POSITION CAMERA 0, 0,0,-10

REM *** Set up particle object ***
LOAD IMAGE "white.bmp",1
MAKE PARTICLES 1,1,1,1

REM *** Increase emissions over a 10 second interval ***
FOR c = 1 TO 10
    SET PARTICLE EMISSIONS 1, c
    WAIT 1000
NEXT c

REM *** End program ***
WAIT KEY
END
```

Activity 41.6

Modify your last program to match that given above and observe the effects produced.

The SET PARTICLE VELOCITY Statement

The velocity of particles produced by the emitter can be modified using the SET PARTICLE VELOCITY statement which has the format shown in FIG-41.13.

FIG-41.13

The SET PARTICLE VELOCITY Statement



In the diagram:

partno

is the integer value previously assigned to the particles object.

value

is an integer value which modifies the velocity of particles produced by the emitter.

This statement affects the height and distance over which the particles travel.

A typical statement might be:

```
SET PARTICLE VELOCITY 1,5
```

Activity 41.7

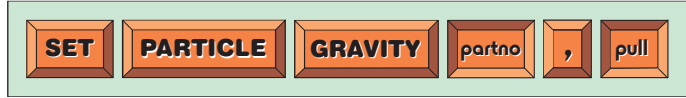
Replace the SET PARTICLE EMISSIONS statement in your previous program with the equivalent SET PARTICLE VELOCITY statement and observe the effect this has on the particles produced.

The SET PARTICLE GRAVITY Statement

In all the examples you've seen so far, particles fly upwards and are then pulled back to some invisible ground level. This pull, or gravity, can be modified using the SET PARTICLE GRAVITY statement which has the format shown in FIG-41.14.

FIG-41.14

The SET PARTICLE GRAVITY Statement



In the diagram:

partno

is the integer value previously assigned to the particles object.

pull

is an integer value which modifies the pull on the particles produced by the emitter. A negative value can be used and this stops the particles falling back towards the emitter.

When a particle object is first created it has a default gravity setting of 5.

This statement affects the height and distance over which the particles travel.

A typical example of this statement might be:

```
SET PARTICLE GRAVITY 1, 5
```

Activity 41.8

Make the following modifications to your last program:

Make the FOR loop start at -5 and finish at 10.

Replace the SET PARTICLE VELOCITY statement with the equivalent SET PARTICLE GRAVITY statement. Test the resulting program.

The SET PARTICLE CHAOS Statement

Although the particles produced from the emitter seem to follow a reasonably predictable path, there is an element of chaotic movement in that path. Just how erratically the particles move can be set using the SET PARTICLE CHAOS statement which has the format shown in FIG-41.15.

FIG-41.15

The SET PARTICLE CHAOS Statement



In the diagram:

partno

is the integer value previously assigned to the particles object.

factor

is an integer value which modifies the chaos factor of the particles produced by the emitter.

When a particles object is first created, it has a default chaos setting of zero.

A typical example of this statement might be:

```
SET PARTICLE CHAOS 1, 10
```

Activity 41.9

In your last program, change the FOR loop to start at -50 and finish at 50.

Replace the SET PARTICLE GRAVITY statement with the equivalent SET PARTICLE CHAOS statement.

Reduce the wait time for each iteration from 1000 to 200.

Test the resulting program.

The SET PARTICLE SPEED Statement

A movie camera takes a series of still photographs on a roll of film. Professional cameras take 25 pictures (known as frames) every second, so about 1/25 of a second passes between one frame being taken and the next. However, some types of movies need a different approach. If we want to film something that happens very slowly, say a flower bud opening, we need to take only one frame of film every minute, but play it back at the normal 25 frames per second.

When handling particles, it is possible to specify how much time has passed between each screen frame using the SET PARTICLES SPEED statement. This will affect the apparent speed of the particles and the number of particles appearing, since some will be created and destroyed in the time between two frames.

This statement takes the format shown in FIG-41.16.

FIG-41.16

The SET PARTICLE
SPEED Statement



In the diagram:

partno is the integer value previously assigned to the particles object.

value is a real value which specifies the seconds that have elapsed between frames.

The default setting for the gap between frames is 0.005 seconds.

A typical statement might be:

```
SET PARTICLE SPEED 1, 0.01
```

The program in LISTING-41.6 demonstrates the effect of this statement by gradually increasing the particles' speed over several seconds.

LISTING-41.6

Changing Particle Speed

```
REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP 0
BACKDROP ON

REM *** Set up camera ***
AUTOCAM OFF
POSITION CAMERA 0, 0,0,-10

REM *** Set up particles object ***
LOAD IMAGE "yellow.bmp",1
MAKE PARTICLES 1,1,1,2
SET PARTICLE VELOCITY 1,0.01

REM *** Increase particles' speed over a 5 second period ***
WAIT KEY
speed#=0.01
FOR c = 1 TO 10
    SET PARTICLE SPEED 1,speed#
    speed# = speed# + 0.005
    WAIT 500
NEXT c

REM *** End program ***
WAIT KEY
END
```

Activity 41.10

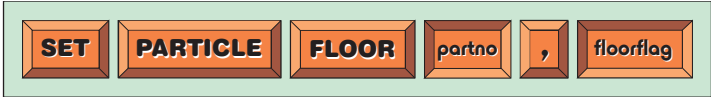
Type in and test the program given in LISTING-41.6 (*particles04.dbpro*).

The SET PARTICLE FLOOR Statement

In all the examples you've seen so far, the particles rise and then fall back onto some invisible ground level. This floor can be removed so that the particles continue to fall until they fade. To do this we use the SET PARTICLE FLOOR statement which takes the format shown in FIG-41.17.

FIG-41.17

The SET PARTICLE FLOOR Statement



In the diagram:

- partno* is the integer value previously assigned to the particles object.
- floorflag* use the value 1 if a floor is required; 0 if no floor is required.

The initial default setting is 1.

This statement effects the height and distance over which the particles travel.

A typical example of this statement might be:

```
SET PARTICLE FLOOR 1,0
```

Activity 41.11

Write a program (*act4111.dbpro*) which creates a particles object, removing its floor when any key is pressed.

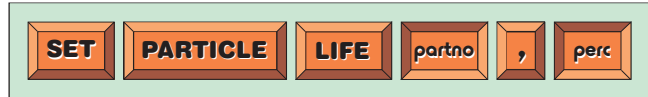
Test the resulting program.

The SET PARTICLE LIFE Statement

After a particle has hit the floor and bounced, it fades away. The time between being created and fading is known as the lifetime of the particle. This lifetime can be modified using the SET PARTICLE LIFE statement which has the format shown in FIG-41.18.

FIG-41.18

The SET PARTICLE LIFE Statement



In the diagram:

partno

is the integer value previously assigned to the particles object.

perc

is an integer value representing the lifetime of particles as a percentage of their normal duration.

Since the normal lifespan of a particle is 100%, we could halve this duration using the statement

```
SET PARTICLE LIFE 1, 50
```

Activity 41.12

In your last program, add a particle life of 20%. This statement should be added immediately after the particles object is created.

Now, test the program.

The GHOST PARTICLES ON Statement

The particles emitted can be made transparent using the GHOST PARTICLES ON statement which has the format shown in FIG-41.19.

FIG-41.19

The GHOST PARTICLE ON Statement



In the diagram:

partno

is the integer value previously assigned to the particles object.

mode

is an integer value representing the transparency mode used by the particles. This can be any value between 0 and 5.

The program shown in LISTING-41.7 creates a particles object in front of a textured cube which helps emphasis the transparency of the particles.

LISTING-41.7

Using Ghost Particles

```
REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP 0
BACKDROP ON

REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 0, 0,0,-10

REM *** Create background cube ***
MAKE OBJECT CUBE 1, 20
LOAD IMAGE "spots.bmp",2
TEXTURE OBJECT 1, 2
POSITION OBJECT 1, 0,0,50

REM *** Set up particles ***
LOAD IMAGE "white.bmp",1
MAKE PARTICLES 1,1,1,2

REM *** Make particles transparent ***
GHOST PARTICLES ON 1,1

REM *** End program ***
WAIT KEY
END
```

Activity 41.13

Type in and test the program given above (*particles05.dbpro*).

Place the GHOST PARTICLES ON statement within a FOR loop using the loop counter (which should range from 0 to 5) as the last value in the GHOST PARTICLES ON statement. Wait for a key press between each value.

The GHOST PARTICLES OFF Statement

The transparency effect created by using GHOST PARTICLES ON can be switched off, thereby returning to normal opaque particles using the GHOST PARTICLES OFF statement which has the format shown in FIG-41.20.

FIG-41.20

The GHOST PARTICLES
OFF Statement



In the diagram:

partno

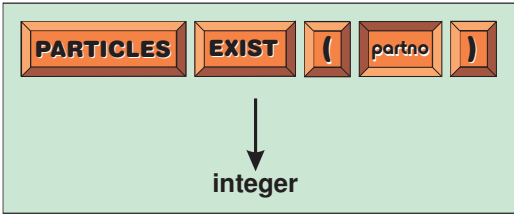
is the integer value previously assigned to the particles object.

Retrieving Data on a Particles Object

The PARTICLES EXIST Statement

To check that a particular particles object has been created, we can use the PARTICLES EXIST statement which has the format shown in FIG-41.21.

FIG-41.21
The **PARTICLES EXIST**
Statement



In the diagram:

partno is an integer value representing the ID of the particles object being tested.

If the specified particles object does exist, then 1 is returned; otherwise zero is returned.

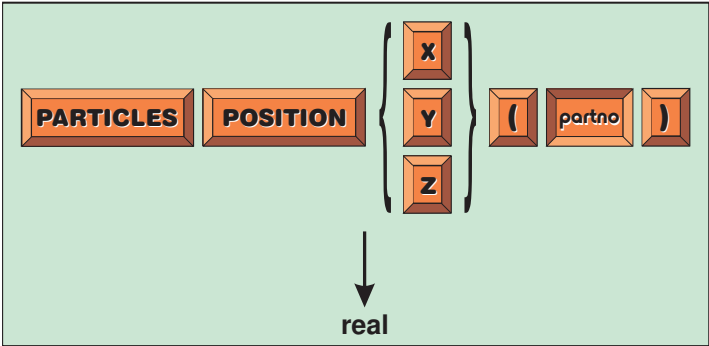
A typical use of this statement might be:

```
IF PARTICLES EXIST(1) = 0
    MAKE PARTICLES 1,1,1,10
ENDIF
```

The PARTICLES POSITION Statement

The exact position of a particles emitter can be determined using the **PARTICLES POSITION** statement. Three variations of the statement exist, with each variation returning one of the object's ordinates. The statement takes the format shown in FIG-41.22.

FIG-41.22
The **PARTICLES**
POSITION Statement



In the diagram:

X,Y,Z One of these options should be chosen. Choose X if the x-ordinate of the specified object is required, Y for the y-ordinate, and Z for the z-ordinate.

partno is the integer value previously assigned to the particles object.

For example, we could determine the position in space of object 1's centre using the lines:

```
x = PARTICLES POSITION X(1)
y = PARTICLES POSITION Y(1)
z = PARTICLES POSITION Z(1)
DO
    SET CURSOR 100,100
    PRINT "Particle emitter at (" ,x," ,",y," ,",z," )"
LOOP
```

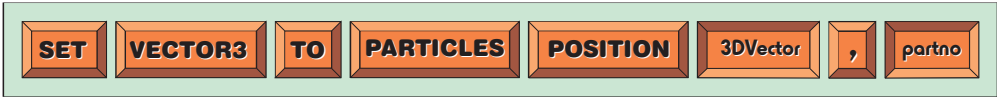
Particles Statements that use Vectors

Two particle-related statements make use of vectors. These are described below.

The SET VECTOR3 TO PARTICLES POSITION Statement

The position of a particles object can be recorded in a vector using the SET VECTOR3 TO PARTICLES POSITION statement which has the format shown in FIG-41.23.

FIG-41.23 The SET VECTOR3 TO PARTICLES POSITION Statement



In the diagram:

<i>3DVector</i>	is an integer value giving the ID of the vector in which the details are saved.
<i>partno</i>	is an integer value giving the ID of the particles object whose position is to be saved.

The SET VECTOR3 TO PARTICLES ROTATION Statement

The angles of rotation of a particles object about its own local axes can be recorded in a vector using the SET VECTOR3 TO PARTICLES ROTATION statement which has the format shown in FIG-41.24.

FIG-41.24 The SET VECTOR3 TO PARTICLES ROTATION Statement



In the diagram:

<i>3DVector</i>	is an integer value giving the ID of the vector in which the details are saved.
<i>partno</i>	is an integer value giving the ID of the particles object whose rotation is to be saved.

Summary

- Using a particles object creates a sparkler effect.
- All particles are projected from a single emitter.
- The colour of the particles emitted is initially determined by the image associated with the particles object.
- Use MAKE PARTICLES to create a particles object.

- Use **HIDE PARTICLES** to make a particles object invisible.
- Use **SHOW PARTICLES** to make a hidden particles object reappear.
- Use **DELETE PARTICLES** to delete a particles object from RAM.
- Use **POSITION PARTICLES** to move the particles object's emitter and currently visible particles.
- Use **POSITION PARTICLE EMISSIONS** to move the emitter, leaving current particles behind.
- Use **ROTATE PARTICLES** to rotate the particles object's emitter about its local axes.
- Use **COLOR PARTICLES** to modify the colour of the particles.
- Use **SET PARTICLE EMISSIONS** to modify the quantity of particles produced.
- Use **SET PARTICLE VELOCITY** to modify the speed at which particles move away from the emitter.
- Use **SET PARTICLE GRAVITY** to affect how particles fall after being emitted.
- Use **SET PARTICLE CHAOS** to affect the random attributes of the particles' paths.
- Use **SET PARTICLE SPEED** to modify the time elapsed between each screen update of the particles' positions.
- Use **SET PARTICLE FLOOR** to disable/enable use of a floor level.
- Use **SET PARTICLE LIFE** to set the life time of the particles.
- Use **GHOST PARTICLES ON** to create transparent particle effects.
- Use **GHOST PARTICLES OFF** to return particles to a normal state after using **GHOST PARTICLES ON**.
- Use **PARTICLES EXIST** to determine if a specific particles object exists.
- Use **PARTICLES POSITION** to determine the coordinates of a particles object's emitter.
- Use **SET VECTOR3 TO PARTICLES POSITION** to save a particles object's coordinates in a vector.
- Use **SET VECTOR3 TO PARTICLES ROTATION** to save a particles object's angles of rotation in a vector.

Other Types of Particles

Introduction

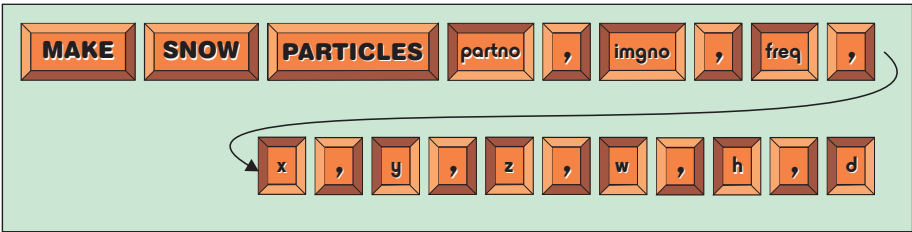
As well as regular particles, DarkBASIC Pro can create two other types of particles effects: snow and fire. Only a few of the regular particles statements can be applied to these types of particles.

The Statements

The MAKE SNOW PARTICLES Statement

Particles can be made to fall like snow from the skies using the MAKE SNOW PARTICLES statement. Rather than originate from a single emitter point, snow particles are created over a rectangular area and fall for a set distance before disappearing. The overall effect is to produce a box of falling particles. The MAKE SNOW PARTICLES statement has the format shown in FIG-41.25.

FIG-41.25
The MAKE SNOW
PARTICLES
Statement



In the diagram:

<i>partno</i>	is an integer value specifying the ID assigned to this particles object.
<i>imgno</i>	is an integer value previously assigned to a loaded image.
<i>freq</i>	is an integer value which modifies the quantity of particles produced.
<i>x,y,z</i>	is a set of real values giving the approximate centre of the area containing the snow particles.
<i>w,h,d</i>	is a set of real values giving the width (<i>w</i>), height (<i>h</i>), and depth (<i>d</i>) of the box in which the snow is to fall.

For example, assuming we have previously loaded an image and assigned it an ID of 2, we could create a snow effect (ID 1) within a 50 x 10 x 20 rectangular area centred on the origin using the line:

```
MAKE SNOW PARTICLES 1,2,100,0,0,0,50,10,20
```

The program in LISTING-41.8 demonstrates the effect of the MAKE SNOW PARTICLES statement.

LISTING-41.8

Using Snow Particles

```
REM *** Set screen resolution and backdrop ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP 0
BACKDROP ON
REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 0, 0,0,-20
REM *** Load image for particles ***
LOAD IMAGE "white.bmp",1
REM *** Create snow ***
MAKE SNOW PARTICLES 1,1,2000,0,0,0,10,10,10
REM *** End program ***
WAIT KEY
END
```

Activity 41.14

Type in and test the program given in LISTING-41.8 (*particles06.dbpro*).

We might reasonably suppose that the position specified in the MAKE SNOW PARTICLES statement ((0,0,0) in the example above) represents the exact centre of the snow box, but, in fact, this is not the case.

Activity 41.15

Modify your last program, placing a 1 unit cube at position (0,0,0). Is the cube at the centre of the snow area?

As you can see, the cube, which has been placed at the same position as that given in the MAKE SNOW PARTICLES statement, is not in the centre of the snow box. In fact, the box is centred in the x and z axes but about 75% of the snow falls below the cube's centre and about 25% above. Remember to take this into account whenever placing snow in your scene.

Some, but not all, of the normal particles statements can be applied to snow particles. For example, snow can be moved to a different position, rotated, shown and hidden, but none of the SET PARTICLE statements operate.

Activity 41.16

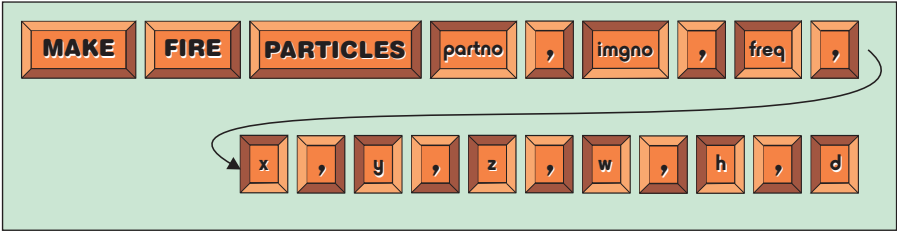
Does the GHOST PARTICLES ON statement affect snow particles?

The MAKE FIRE PARTICLES Statement

The MAKE FIRE PARTICLES statement creates the effect of sparks rising gently from an unseen fire. In reality it is the same effect as the MAKE SNOW PARTICLES statement but with the particles travelling upwards rather than down! The statement has the format shown in FIG-41.26.

FIG-41.26

The MAKE FIRE PARTICLES Statement



In the diagram:

<i>partno</i>	is an integer value specifying the ID assigned to this particles object.
<i>imgno</i>	is an integer value previously assigned to a loaded image.
<i>freq</i>	is an integer value which modifies the quantity of particles produced.
<i>x,y,z</i>	is a set of real values giving the approximate centre of the area containing the fire particles.
<i>w,h,d</i>	is a set of real values giving the width (<i>w</i>), height (<i>h</i>), and depth (<i>d</i>) of the box in which the fire particles are to rise.

Activity 41.17

Modify *particles06.dbpro* to show fire particles rather than snow.

(Use the image *red.bmp* rather than *white.bmp*.)

The same restrictions apply to fire particles as snow particles with many of the regular particles statements not operating on these particles objects.

Summary

- Use MAKE SNOW PARTICLES to create a snowing effect.
- The box in which the snow appears is based around the position (*x*, *y*, *z*) but, although centred on the *x* and *z* values, is about 75% below the *y* value.
- Use MAKE FIRE PARTICLES to create rising spark effects.
- Only some of the standard particles object statements can be applied to snow and fire particles.

Examples of Using Particles

Introduction

We can combine particles with other 3D elements to create some visually interesting effects. A few possibilities are described below.

A Roman Candle

All we need is a cylinder and a particles object to create a roman candle firework (see FIG-41.27) as coded in the program given in LISTING-41.9.

LISTING-41.9

A Roman Candle

```
REM *** Define constants ***
REM *** Image IDs ***
#CONSTANT YellowImg      1
#CONSTANT WrapperImg     2

REM *** Object IDs ***
#CONSTANT FireworkObj    1
REM *** Particle IDs ***
#CONSTANT YellowPart     1

REM *** Set screen resolution and backdrop ***
SET DISPLAY MODE 800,600,32
COLOR BACKDROP 0
BACKDROP ON

REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 0, 0,3,-20

REM *** Load images ***
LOAD IMAGE "yellow.bmp",YellowImg
LOAD IMAGE "firework.bmp",WrapperImg

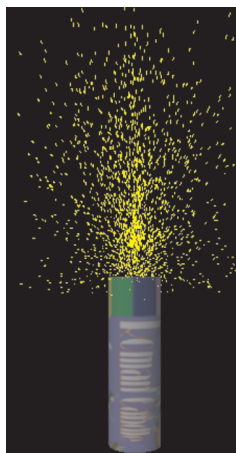
REM *** Create firework ***
MAKE OBJECT CYLINDER FireworkObj,2
TEXTURE OBJECT FireworkObj,WrapperImg
SCALE OBJECT FireworkObj, 100,300,100

REM *** Create particles ***
MAKE PARTICLES YellowPart,YellowImg,20,1
SET PARTICLE GRAVITY YellowPart,0.1
POSITION PARTICLES YellowPart, 0,3,0
SET PARTICLE FLOOR YellowPart, 0

REM *** End program ***
WAIT KEY
END
```

FIG-41.27

A Roman Candle



Activity 41.18

Type in and test the program in LISTING-41.9 (*particles07.dbpro*).

Modify the program so that the firework emits both red and yellow particles. (HINT: Add a second particles object and use the image *red.bmp*.)

A Spaceship

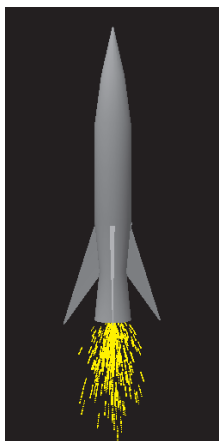
Here's a good old-fashioned 1950s Sci-Fi spaceship just about to take off (see FIG-41.28 and LISTING-41.10)

LISTING-41.10

A Spaceship Launch

FIG-41.28

Spacecraft Launching



```
REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP 0
BACKDROP ON

REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 0, 0,0,-20
POINT CAMERA 0,0,0

REM *** Load spaceship ***
LOAD OBJECT "spaceship2.x",1

REM *** Load image for particles ***
LOAD IMAGE "yellow.bmp",1
REM *** Create particles object ***
MAKE PARTICLES 1,1,1,1.5
POSITION PARTICLES 1,0,-3.6,0
ROTATE PARTICLES 1, 0,0,180
SET PARTICLE EMISSIONS 1, 20
SET PARTICLE VELOCITY 1, 5
SET PARTICLE GRAVITY 1, -4
SET PARTICLE SPEED 1,0.02
SET PARTICLE LIFE 1,10
REM *** End program ***
WAIT KEY
END
```

Activity 41.19

Type in and test the program in LISTING-41.10 (*particles08.dbpro*).

Modify the program so that the spaceship's engine blast only appears after a key press. The rocket should then actually take off, moving out of the top of the screen. Make the rocket also move in a z direction as it lifts - this will help give the illusion of it moving away from the viewer.

A Dungeon Torch

It's a well-known fact that a proper dungeon always uses flaming torches to light their long dank corridors. By combining particles and a light map, we can recreate an illusion of that setup in DarkBASIC Pro.

The torch itself is just a cone and a particle emitter. A point light is placed at approximately the same position as the particles. The light map used back in Chapter 34 is used to create a circle of light on the wall. Finally, to create a flickering effect the light's range continually changes (see FIG-41.29 and LISTING-41.11).

LISTING-41.11

A Dungeon Torch

```
SetUpScreen ()
CreateScene ()
DO
    SET LIGHT RANGE 1, RND(1000)
    WAIT 10
LOOP
```

continued on next page

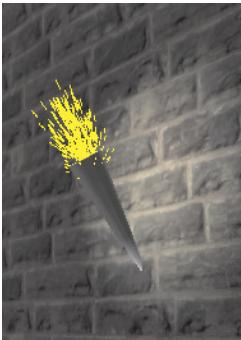
LISTING-41.11

(continued)

A Dungeon Torch

FIG-41.29

A Flaming Torch



```
WAIT KEY
```

```
END
```

```
FUNCTION SetUpScreen()
```

```
    SET DISPLAY MODE 1280,1024,32
```

```
    COLOR BACKDROP RGB(20,20,20)
```

```
    BACKDROP ON
```

```
    AUTOCAM OFF
```

```
    POSITION CAMERA 20,-4,-5
```

```
    POINT CAMERA 0,0,3
```

```
ENDFUNCTION
```

```
FUNCTION CreateScene()
```

```
    REM *** Create wall ***
```

```
    MAKE OBJECT PLAIN 2,30,20
```

```
    LOAD IMAGE "brickslarge.jpg",2
```

```
    TEXTURE OBJECT 2,2
```

```
    POSITION OBJECT 2,0,-1,5
```

```
    REM *** Create torch holder ***
```

```
    MAKE OBJECT CONE 3,3
```

```
    SCALE OBJECT 3,25,100,25
```

```
    ZROTATE OBJECT 3,180
```

```
    XROTATE OBJECT 3, 30
```

```
    POSITION OBJECT 3, 0,-2,4.5
```

```
    REM *** Create flames ***
```

```
    LOAD IMAGE "yellow.bmp",3
```

```
    MAKE PARTICLES 1,3,1,1
```

```
    POSITION PARTICLES 1, 0,-1,4
```

```
    ROTATE PARTICLES 1,-30,0,0
```

```
    SET PARTICLE VELOCITY 1, 5
```

```
    SET PARTICLE GRAVITY 1,-1
```

```
    SET PARTICLE LIFE 1,10
```

```
    SET PARTICLE SPEED 1,0.002
```

```
    REM *** Put light in sphere ***
```

```
    MAKE LIGHT 1
```

```
    SET POINT LIGHT 1,0,1,2
```

```
    SET LIGHT RANGE 1,1000
```

```
    REM *** Add Light maps ***
```

```
    LOAD IMAGE "lightmap.jpg",4
```

```
    SET LIGHT MAPPING ON 2,4
```

```
    LOAD IMAGE "lampmap.jpg",5
```

```
    SET LIGHT MAPPING ON 3,5
```

```
ENDFUNCTION
```

Activity 41.20

Type in and test the program in LISTING-41.11 (*particles09.dbpro*).

Solutions

Activity 41.1

The particles become much larger with the new setting.

Activity 41.2

```
REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP 0
BACKDROP ON
REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 0,0,5,-10
REM *** Load image for particles ***
LOAD IMAGE "green.bmp",1
REM *** Create a particles object ***
MAKE PARTICLES 1,1,1,10
REM *** Remove after key press ***
WAIT KEY
HIDE PARTICLES 1
REM *** End program ***
WAIT KEY
END
```

Activity 41.3

```
REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP 0
BACKDROP ON
REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 0, 0,0,-16
REM *** Load image for particles ***
LOAD IMAGE "yellow.bmp",1
REM *** Create particles object ***
MAKE PARTICLES 1,1,1,1
REM *** Move particle object ***
post# = -10
FOR c = 1 TO 200
    POSITION PARTICLES 1,0,post#,0
    post# = post# + 0.1
    WAIT 10
NEXT c
REM *** End program ***
WAIT KEY
END
```

Activity 41.4

No solution required.

Activity 41.5

No solution required.

Activity 41.6

No solution required.

Activity 41.7

```
REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP 0
BACKDROP ON
REM *** Set up camera ***
AUTOCAM OFF
POSITION CAMERA 0, 0,0,-10
```

```
REM *** Set up particle object ***
LOAD IMAGE "white.bmp",1
MAKE PARTICLES 1,1,1,1
REM *** Increase velocity over 10 secs***
FOR c = 1 TO 10
    SET PARTICLE VELOCITY 1, c
    WAIT 1000
NEXT c
REM *** End program ***
WAIT KEY
END
```

Activity 41.8

```
REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP 0
BACKDROP ON
REM *** Set up camera ***
AUTOCAM OFF
POSITION CAMERA 0, 0,0,-10
REM *** Set up particle object ***
LOAD IMAGE "white.bmp",1
MAKE PARTICLES 1,1,1,1
REM *** Increase gravity 15 second ***
FOR c = -5 TO 10
    SET PARTICLE GRAVITY 1, c
    WAIT 1000
NEXT c
REM *** End program ***
WAIT KEY
END
```

Activity 41.9

```
REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP 0
BACKDROP ON
REM *** Set up camera ***
AUTOCAM OFF
POSITION CAMERA 0, 0,0,-10
REM *** Set up particle object ***
LOAD IMAGE "white.bmp",1
MAKE PARTICLES 1,1,1,1
REM *** Increase chaos ***
FOR c = -50 TO 50
    SET PARTICLE CHAOS 1, c
    WAIT 200
NEXT c
REM *** End program ***
WAIT KEY
END
```

Activity 41.10

No solution required.

Activity 41.11

```
REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP 0
BACKDROP ON
REM *** Set up camera ***
AUTOCAM OFF
POSITION CAMERA 0, 0,0,-10
REM *** Set up particle object ***
LOAD IMAGE "yellow.bmp",1
MAKE PARTICLES 1,1,1,2
SET PARTICLE GRAVITY 1,0.5
```

```

REM *** Remove floor after a key press ***
WAIT KEY
SET PARTICLE FLOOR 1,0
REM *** End program ***
WAIT KEY
END

```

```

WAIT KEY
END

```

As you can see from the result, the cube is within the snow-bound area. It is centred in the x direction (and z) but about three-quarters of the way up in the y direction.

Activity 41.12

```

REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP 0
BACKDROP ON
REM *** Set up camera ***
AUTOCAM OFF
POSITION CAMERA 0, 0,0,-10
REM *** Set up particle object ***
LOAD IMAGE "yellow.bmp",1
MAKE PARTICLES 1,1,1,2
SET PARTICLE LIFE 1, 20
SET PARTICLE GRAVITY 1,0.5
REM *** Remove floor after a key press ***
WAIT KEY
SET PARTICLE FLOOR 1,0
REM *** End program ***
WAIT KEY
END

```

Activity 41.16

GHOST PARTICLES ON does not affect the snow particles.

Activity 41.17

```

REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP 0
BACKDROP ON
REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 0, 0,0,-20
REM *** Load image for particles ***
LOAD IMAGE "red.bmp",1
REM *** Create fire ***
MAKE FIRE PARTICLES 1,1,2000,0,0,0,10,10,10
REM *** End program ***
WAIT KEY
END

```

Activity 41.13

```

REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP 0
BACKDROP ON
REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 0,0,0,-10
REM *** Set up particles ***
LOAD IMAGE "yellow.bmp",1
MAKE PARTICLES 1,1,1,2
REM *** Create background cube ***
MAKE OBJECT CUBE 1,20
LOAD IMAGE "spot1.bmp",2
TEXTURE OBJECT 1,2
POSITION OBJECT 1,0,0,50
REM *** Test each ghost option ***
FOR c = 0 TO 5
  GHOST PARTICLES ON 1,c
  WAIT KEY
NEXT c
REM *** End program ***
WAIT KEY
END

```

Activity 41.18

The second version of the firework is coded as:

```

REM *** Define constants ***
REM *** Image IDs ***
#CONSTANT YellowImg 1
#CONSTANT WrapperImg 2
#CONSTANT RedImg 3
REM *** Object IDs ***
#CONSTANT FireworkObj 1
REM *** Particle IDs ***
#CONSTANT YellowPart 1
#CONSTANT RedPart 2
REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP 0
BACKDROP ON
REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 0, 0,3,-20
REM *** Load images ***
LOAD IMAGE "yellow.bmp",YellowImg
LOAD IMAGE "firework.bmp",WrapperImg
LOAD IMAGE "red.bmp",RedImg
REM *** Create firework ***
MAKE OBJECT CYLINDER FireworkObj,2
TEXTURE OBJECT FireworkObj,WrapperImg
SCALE OBJECT FireworkObj, 100,300,100
REM *** Create yellow particles ***
MAKE PARTICLES YellowPart,YellowImg,20,1
POSITION PARTICLES YellowPart, 0,3,0
SET PARTICLE GRAVITY YellowPart,0.1
SET PARTICLE FLOOR YellowPart, 0
REM *** Create red particles ***
MAKE PARTICLES RedPart,RedImg,20,1
POSITION PARTICLES RedPart, 0,3,0
SET PARTICLE GRAVITY RedPart,0.1
SET PARTICLE FLOOR RedPart,0
REM *** End program ***
WAIT KEY
END

```

Activity 41.14

No solution required.

Activity 41.15

```

REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP 0
BACKDROP ON
REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 0, 0,0,-20
REM *** Load image for particles ***
LOAD IMAGE "white.bmp",1
REM *** Create snow ***
MAKE SNOW PARTICLES 1,1,2000,0,0,0,10,10,10
REM *** Create cube ***
MAKE OBJECT CUBE 1,1
REM *** End program ***

```

Activity 41.19

The second version of the program is coded as:

```
REM *** Set up screen ***
SET DISPLAY MODE 1280,1024,32
COLOR BACKDROP 0
BACKDROP ON
REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 0, 0,0,-40
POINT CAMERA 0,10,0
LOAD OBJECT "spaceship2.x",1
REM *** Load image for particles ***
LOAD IMAGE "yellow.bmp",1
REM *** Create particles object ***
MAKE PARTICLES 1,1,1,1.5
POSITION PARTICLES 1,0,-3.6,0
ROTATE PARTICLES 1, 0,0,180
SET PARTICLE EMISSIONS 1, 20
SET PARTICLE VELOCITY 1, 5
SET PARTICLE GRAVITY 1, -4
SET PARTICLE SPEED 1,0.02
SET PARTICLE LIFE 1,10
REM *** Press key to launch ***
HIDE PARTICLES 1
WAIT KEY
SHOW PARTICLES 1
z# = 0
DO
    POSITION OBJECT 1,0,OBJECT POSITION Y(1)
    +1, z#
    POSITION PARTICLES 1,0,OBJECT POSITION
Y(1)
    -3.6,z#
    z# = z# +0.5
    WAIT 20
LOOP
REM *** End program ***
END
```

Activity 41.20

No solution required.

The Elevators Game

Correcting Logic Errors

Creating a 3rd Person Perspective Game

Using Previously Created Models in a Game

Elevators

Introduction

This started out as a snakes-and-ladders game until it became apparent that creating the snakes and getting the characters to climb ladders was going to prove to be too time-consuming. So we now have elevators in place of both the snakes and the ladders. Some elevators take you up, others bring you down.

The Equipment

The game consists of a board showing numbered squares, the player's character and a dice. The board squares are laid out in horizontal rows, with 10 squares per row. The squares are numbered sequentially from 1 to 100. Also on the board are a set of elevators used to carry the player from one level to another.

The Aim

The aim of the game is to get the player's character from square 1 to square 100.

The Rules

The basic rules come from the snakes and ladders game. The player starts on square 1 and moves forward according to the value thrown on the dice. If, at the end of a move, the character is on an elevator platform, it takes that elevator to the destination square - sometimes upwards; sometimes down.

Creating a Computer version of the Game

User Controls

There is little interaction; the player needs only to press a key to cause the dice to roll. The character is then moved automatically.

Game Responses

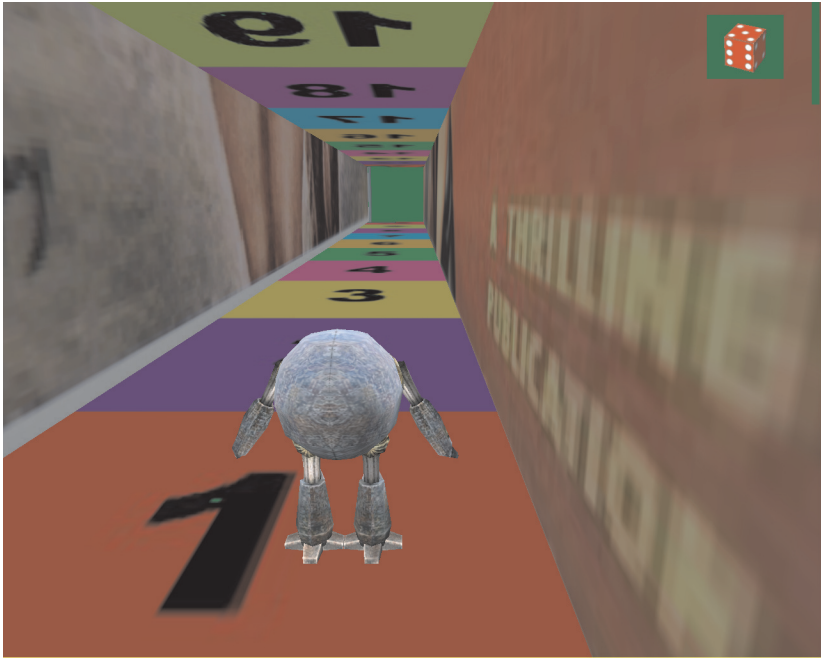
The game reacts to a dice throw by moving the player's character by the value thrown. If the character has to move from one row of the board to the next during a move, this is done using the lift at the end of the row. If the character stops on an elevator square it is carried off to that elevator's destination.

Screen Layout

To create an unique look to the game, it is played in 3D with each row of the board becoming a corridor down which the player's character is to move. The camera view gives a 3rd person perspective, the scene being shown from a viewpoint just behind the character's shoulder (see FIG-42.1).

FIG-42.1

A Screenshot from the Game



The Board Design

We have to begin by designing the basic board, showing the elevators and their destinations. This can be done in an abstract way (see FIG-42.2).

FIG-42.2

The Board Layout
Showing the Elevators

100	99	98	97	96	95	94	93	92	91
81	82	83	84	85	86	87	88	89	90
80	79	78	77	76	75	74	73	72	71
61	62	63	64	65	66	67	68	69	70
60	59	58	57	56	55	54	53	52	51
41	42	43	44	45	46	47	48	49	50
40	39	38	37	36	35	34	33	32	31
21	22	23	24	25	26	27	28	29	30
20	19	18	17	16	15	14	13	12	11
1	2	3	4	5	6	7	8	9	10

Notice that all the lifts are vertical. Anything lying at an angle would just cause more modelling problems.

The Media Used

The following 3D objects are required:

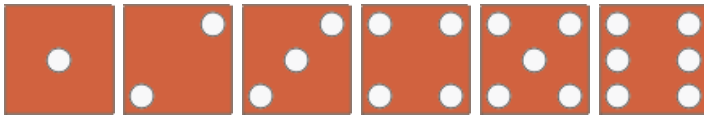
- Board (created as a tall, narrow box with 9 internal floors)
- Elevator

- Player's character
- Dice

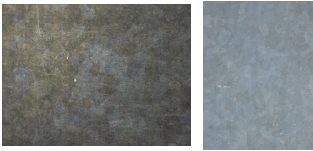
The elevator and dice models were created in Chapter 35; the player's character model and its texture have been taken from the *The Game Creators'* Dark Matter 1 model pack and the board was constructed using *chUmbaLum sOf*t's Milkshape modelling program.

The other images required are the textures for the 3D objects and are shown in FIG-42.3.

FIG-42.3 The Images used to Texture 3D Objects



Dice images



Elevator Images



Board Images

Data Structures

We need to set up a record structure to hold details about a single elevator. The information required is:

- the elevator object's ID number
- the square on which the lift starts
- the square on which the lift terminates
- the number of floors covered by the lift (this will be negative for descending elevators)
- the z-ordinate of the lift's position on a square

All this can be held in a record structure:

```

TYPE ElevatorType
  ID AS INTEGER
  startsquare AS INTEGER

```

```

        endsquare AS INTEGER
        floors AS INTEGER `(-ve for downwards)
        zord AS INTEGER
    ENDTYPE

```

There are 27 elevators in total, so we need an array to store information about each:

```

    GLOBAL DIM lifts(27) AS ElevatorType

```

The board itself need only record, for each square, the presence of an elevator starting on that square:

```

    GLOBAL DIM board(100) AS INTEGER

```

All we need to know about the player's character is its position on the board:

```

    GLOBAL playerat AS INTEGER

```

The 3D objects used in the game have already been textured and only need to be assigned ID values within the game. As a general rule we should assign a name to each ID, but since there are 27 elevators we'll assign a name to the first one only.

```

    REM *** Object constants ***
    #CONSTANT boardobj          1
    #CONSTANT playerobj         2
    #CONSTANT diceobj           3
    #CONSTANT firstliftobj      4 `27 lifts with IDs 4 - 30

```

So, the final code for our main data and constants is:

```

    REM *****
    REM ***      Structures      ***
    REM *****
    REM *** Elevator details ***
    TYPE ElevatorType
        ID AS INTEGER
        startsquare AS INTEGER
        endsquare AS INTEGER
        floors AS INTEGER `(-ve for downwards)
        zord AS INTEGER
    ENDTYPE
    REM *****
    REM ***      Global variables      ***
    REM *****
    GLOBAL DIM Lifts(27) AS ElevatorType
    GLOBAL DIM board(100) AS INTEGER
    GLOBAL playerat AS INTEGER
    REM *****
    REM ***      Constants      ***
    REM *****
    REM *** Object constants ***
    #CONSTANT boardobj          1
    #CONSTANT playerobj         2
    #CONSTANT diceobj           3
    #CONSTANT firstliftobj      4 `27 lifts with IDs 4 - 30

```

Game Logic

As usual, we want to describe the whole logic of the program in less than a dozen lines and then look at each of these sections in more detail until we have our final product. So, the overall game could be described as follows:

```

Set up the game
REPEAT
    Roll the dice
    Move the player
UNTIL the player reaches square 100
Show finish screen

```

As usual, the descriptive lines become function calls in the actual program. We'll need to find out what value is thrown on the dice so it can be used when moving the player, so the function for throwing the dice must return the dice value and the function which moves the player must take that value as a parameter. This gives us the following code for the main section:

```

REM *****
REM ***          Main game logic          ***
REM *****
SetUpGame()
REPEAT
    thrown = RollDice()
    MovePlayer(thrown)
UNTIL playerat = 100
EndScreen()
END

```

Activity 42.1

Create a new project (*elevators.dbpro*) and copy into it the code already given for the data structure, global variables and constants as well as the logic of the main section.

At the end of the code, add a REM statement containing the lines:

```

REM *****
REM ***          Level 1 routines          ***
REM *****

```

Under these REM statements, write test stubs for each function called. The test stubs should be empty, containing only FUNCTION and ENDFUNCTION statements.

The test stub for *RollDice()* should return the value 1.

Run the program and make sure it executes. *You'll have to press Escape to exit the REPEAT..UNTIL structure in the main section.*

Adding *SetUpGame()*

There are many tasks to be performed by this part of the program, but we can split those tasks into two main sections:

- Initialisation of the data
- Initialisation of the visuals

So we can code *SetUpGame()* as a function which does nothing more than call two other functions:

```

FUNCTION SetUpGame()
    InitialiseData()
    InitialiseVisuals()
ENDFUNCTION

```

LISTING-42.1

Elevators - Getting
Started with the Code

These new routines will then require their own test stubs.

The complete program at this point is shown in LISTING-42.1.

```
REM *****
REM ***      Structures      ***
REM *****
REM *** Elevator details ***
TYPE ElevatorType
  ID AS INTEGER
  startsquare AS INTEGER
  endsquare AS INTEGER
  floors AS INTEGER `(-ve for downwards)
  zord AS INTEGER
ENDTYPE
REM *****
REM ***      Global variables      ***
REM *****
GLOBAL DIM lifts(27) AS ElevatorType
GLOBAL DIM board(100) AS INTEGER
GLOBAL playerat AS INTEGER
REM *****
REM ***      Constants      ***
REM *****
REM *** Object constants ***
#CONSTANT boardobj      1
#CONSTANT playerobj     2
#CONSTANT diceobj       3
#CONSTANT firstliftobj  4 `27 lifts IDs 4 - 30

REM *****
REM ***      Main game logic      ***
REM *****
SetUpGame()
REPEAT
  thrown = RollDice()
  MovePlayer(thrown)
UNTIL playerat = 100
EndScreen()
END

REM *****
REM ***      Level 1 routines      ***
REM *****
FUNCTION SetUpGame()
  InitialiseData()
  InitialiseVisuals()
ENDFUNCTION

FUNCTION RollDice()
ENDFUNCTION 1

FUNCTION MovePlayer(thrown)
ENDFUNCTION

FUNCTION EndScreen()
ENDFUNCTION

REM *****
REM ***      Level 2 routines      ***
REM *****
FUNCTION InitialiseData()
ENDFUNCTION

FUNCTION InitialiseVisuals()
ENDFUNCTION
```

Notice how the functions have been grouped; a function called by the main section is a level 1 routine; a routine called by a level 1 function is a level 2 routine. Large programs may have levels 3, 4 or more. This is a popular layout method for many programmers.

Activity 42.2

Modify your program to match the code above.

Adding *InitialiseData()*

The main tasks of the *InitialiseData()* function involves us setting up the contents of the *lifts* and *board* arrays, but we'll also need to specify the player's starting position and seed the random number generator. So the routine can be coded as:

```
FUNCTION InitialiseData()
    InitialiseLifts()
    InitialiseBoard()
    REM *** Start player at square 1 ***
    playerat = 1
    REM *** Seed random number generator***
    RANDOMIZE TIMER()
ENDFUNCTION
```

Activity 42.3

Replace the *InitialiseData()* stub with the code given.

Add test stubs for the two new routines in a level 3 section of the program.

Adding *InitialiseLifts()*

We need a great deal of data to initialise the *lifts* array since we have to set up information about each lift. The easiest way to set these details up is to use DATA statements. These values can then be read into the appropriate part of the array.

The required coding is given below:

```
FUNCTION InitialiseLifts()
    REM *** Lifts start at square ***
    DATA 10,13,18,20,24,25,30,34,35
    DATA 39,40,49,50,53,54,59,60,65
    DATA 68,70,72,75,80,83,87,90,97
    REM *** Lifts end at square ***
    DATA 11,33,43,21,64,56,31,27,55
    DATA 19,41,12,51,48,67,79,61,76
    DATA 88,71,52,95,81,58,74,91,77
    REM *** Floors covered by lifts ***
    DATA 1,2,3,1,4,3,1,-1,2
    DATA -2,1,-3,1,-1,1,2,1,1
    DATA 2,1,-2,2,1,-3,-1,1,-2
    REM *** z-ordinates of lifts' position ***
    DATA 47,23,-27,-47,-17,-7,47,13,3
    DATA -37,-47,33,47,23,13,-37,-47,-7
    DATA 23,47,33,3,-47,-27,13,47,-17
    REM *** Store lift IDs (firstliftobj to firstliftobj + 26)***
    FOR c = 1 TO 27
        lifts(c).ID = firstliftobj + c-1
    NEXT c
```



```

    REM *** Store start square for lift ***
    FOR c = 1 TO 27
        READ sq
        lifts(c).startsquare = sq
    NEXT c
    REM *** Store end square for lift ***
    FOR c = 1 TO 27
        READ fin
        lifts(c).endsquare = fin
    NEXT c
    REM *** Store floors covered ***
    FOR c = 1 TO 27
        READ fl
        lifts(c).floors = fl
    NEXT c
    REM *** Store z-ord of lift ***
    FOR c = 1 TO 27
        READ z
        lifts(c).zord = z
    NEXT c
ENDFUNCTION

```

Adding *InitialiseBoard()*

In the *board* array, each cell represents one square on the board. If a square does not contain the starting position of an elevator, we store zero in the corresponding element of *board*; if a square does contain an elevator, then the corresponding cell in *board* contains the elevator's number.

We have two choices when storing an elevator number; we can either store the elevator's ID or we can store the position of that elevator's details in the *lifts* array.

For example, square 10 on the board has an elevator (ID 4) taking the player to square 11, so we could either write

```
board(10) = 4
```

thereby storing the elevator's ID, or we could use

```
board(10) = 1
```

meaning that the elevator starting at square 10 has its details stored in *lifts(1)*.

In fact, the second option is the better approach since it allows more flexibility should we ever need to change the ID values used for the elevator objects.

As a result, the code for the *InitialiseBoard()* routine is:

```

FUNCTION InitialiseBoard()
    REM ***Square empty (0) or elevator's location in lifts ***
    DATA -1,-1,-1,-1,-1,  -1,-1,-1,-1,1
    DATA -1,-1,2,-1,-1,  -1,-1,3,-1,4
    DATA -1,-1,-1,5,6,  -1,-1,-1,-1,7
    DATA -1,-1,-1,8,9,  -1,-1,-1,10,11
    DATA -1,-1,-1,-1,-1, -1,-1,-1,12,13
    DATA -1,-1,14,15,-1, -1,-1,-1,16,17
    DATA -1,-1,-1,-1,18, -1,-1,19,-1,20
    DATA -1,21,-1,-1,22, -1,-1,-1,-1,23
    DATA -1,-1,24,-1,-1, -1,25,-1,-1,26
    DATA -1,-1,-1,-1,-1, -1,27,-1,-1,-1
    REM *** Store this data in board array ***
    FOR c = 1 TO 100
        READ board(c)
    
```

The rather strange code needs explaining; storing 0's in the DATA statements seemed to upset the program and so it was necessary to store -1's and convert them to zeros!

```

        IF board(c) = -1
            board(c) = 0
        ENDIF
    NEXT c
ENDFUNCTION

```

Activity 42.4

Replace the stubs for *InitialiseLifts()* and *InitialiseBoard()* with the code given.

Adding *InitialiseVisuals()*

Now we're ready to load all the visual elements of the game and position the cameras. The requirements of this routine can best be described as:

```

Set up the screen
Load the board
Load the player's character
Load the dice
Position the cameras
Set the lighting

```

All but the first and last of these tasks will involve a few lines of code, and it is probably best to create a set of level 3 routines to handle the details. So, *InitialiseVisuals()* can be coded as:

```

FUNCTION InitialiseVisuals()
    REM *** Set the screen resolution ***
    SET DISPLAY MODE 1280,1024,32
    LoadBoard()
    LoadPlayerCharacter()
    LoadDice()
    PositionCameras()
    REM *** Increase the ambient light ***
    SET AMBIENT LIGHT 50
ENDFUNCTION

```

The standard screen resolution is used and the ambient light is turned up since the image was too dark with the default setting.

Activity 42.5

Replace the stub of *InitialiseVisuals()* with the code given above and create new level 3 stubs for the functions it calls.

Loading Models and Texture Files

Before continuing, it would be better if we copied the necessary model and image files to the current directory. The following files need to be copied:

<i>board.x</i>	the model of the board
<i>dice.dbo</i>	the model of the dice
<i>H-Android_Move.x</i>	the player's character model
<i>lift.dbo</i>	the elevator model

<i>cherry.jpg</i>	a board texture
<i>bbrush.jpg</i>	a board texture
<i>floor1.bmp</i>	a board texture
<i>floor2.bmp</i>	a board texture
<i>floor3.bmp</i>	a board texture

<i>floor4.bmp</i>	a board texture
<i>floor5.bmp</i>	a board texture
<i>floor6.bmp</i>	a board texture
<i>floor7.bmp</i>	a board texture
<i>floor8.bmp</i>	a board texture
<i>floor9.bmp</i>	a board texture
<i>floorA.bmp</i>	a board texture
<i>spot1.jpg</i>	a dice texture
<i>spot2.jpg</i>	a dice texture
<i>spot3.jpg</i>	a dice texture
<i>spot4.jpg</i>	a dice texture
<i>spot5.jpg</i>	a dice texture
<i>spot6.jpg</i>	a dice texture
<i>robot.dds</i>	a player character texture
<i>metal.bmp</i>	an elevator texture
<i>steel.bmp</i>	an elevator texture

All files used in this book
are available from our
website -
www.digital-skills.co.uk

Activity 42.6

Copy the necessary files into your folder.

Adding *LoadBoard()*

There are actually two parts to this routine: first we need to load and resize the board model, but also we need to position the elevators within the board. This second task will be performed by a separate routine. The code for *LoadBoard()* is:

```

FUNCTION LoadBoard()
    REM *** Load, scale and position board ***
    LOAD OBJECT "board.x",1
    SCALE OBJECT 1, 1000,1000,1000
    YROTATE OBJECT 1, 90
    POSITION OBJECT 1,5,50,0
    REM *** Make black areas invisible ***
    REM *** These are the holes in the floor ***
    REM *** through which the lifts pass ***
    SET OBJECT TRANSPARENCY 1,1
    AddElevators()
ENDFUNCTION

```

Adding *AddElevators()*

This routine loads the elevator model (*lift.dbo*) and creates another 26 clones of the model. The data in the *board* and *lifts* arrays is then used to position and resize each elevator.

```

FUNCTION AddElevators()
    REM *** Create the first lift ***
    LOAD OBJECT "lift.dbo",firstliftobj
    REM *** Create others as clones ***
    FOR c = 2 TO 27
        CLONE OBJECT lifts(c).ID,firstliftobj
    NEXT c
    REM *** Position all the lifts ***
    FOR c = 1 TO 27
        REM *** Rotate lift and scale to match floors ***
        liftID = lifts(c).ID
        YROTATE OBJECT liftID,-90
    NEXT c
ENDFUNCTION

```

```

SCALE OBJECT liftID,100,10*ABS(lifts(c).floors),100
REM *** position up lifts at start square ***
IF lifts(c).floors> 0
    POSITION OBJECT liftID,-4.5,
    ↵ (lifts(c).startsquare-1)/10*10,lifts(c).zord
ELSE
    REM *** And down ones at the finish square ***
    POSITION OBJECT liftID,-4.5,
    ↵ (lifts(c).endsquare-1)/10*10,lifts(c).zord
    REM *** down lifts need platform at top ***
    OFFSET LIMB liftID,2,0,100,-2
ENDIF
NEXT c
ENDFUNCTION

```

Since this is called by a level 3 function, it needs to be placed in a level 4 section.

Adding *LoadPlayerCharacter()*

This routine needs to load the appropriate model, scale and rotate it. The code is:

```

FUNCTION LoadPlayerCharacter()
    REM *** Load, scale and position player's character ***
    LOAD OBJECT "H-android-move.x",playerobj
    SCALE OBJECT playerobj,150,150,150
    YROTATE OBJECT playerobj,180
    POSITION OBJECT playerobj,2,0,-45 `square 1
    REM *** Set the animation play speed ***
    SET OBJECT SPEED playerobj,15
ENDFUNCTION

```

Notice that the routine takes the trouble to check the position of the player on the board and rotates the character accordingly. This may seem unnecessary since the player will obviously start at square 1 - however, while testing the program we may want to start the player at other positions.

Adding *LoadDice()*

Like *LoadPlayerCharacter()*, *LoadDice()* involves loading and scaling a model:

```

FUNCTION LoadDice()
    REM *** Load, scale and position dice ***
    LOAD OBJECT "dice.dbo",diceobj
    POSITION OBJECT diceobj,30,15,45
    SCALE OBJECT diceobj,50,50,50
ENDFUNCTION

```

Adding *PositionCameras()*

Two cameras are used in the game. Camera 1 follows the player's character as it moves through the corridors of the board and forms the main image on the screen; camera 2 points at the dice and its output is shown as a picture-in-picture window at the top right of the screen. The code for this routine is:

```

FUNCTION PositionCameras()
    AUTOCAM OFF
    REM *** Set up player's camera ***
    MAKE CAMERA 1
    SET CAMERA VIEW 1, 0,0,1279,1023
    POSITION CAMERA 1,3,5,-52
    POINT CAMERA 1,3,2,-40
    REM *** Set up dice's camera ***

```

```

MAKE CAMERA 2
POSITION CAMERA 2,50,43,0
POINT CAMERA 2,30,15,45
SET CAMERA FOV 2, 10
SET CAMERA VIEW 2,1100,20,1220,120
ENDFUNCTION

```

At this point, our program is as shown in LISTING-42.2.

LISTING-42.2

The Game So Far

```

REM *****
REM ***      Structures      ***
REM *****
REM *** Elevator details ***
TYPE ElevatorType
  ID AS INTEGER
  startsquare AS INTEGER
  endsquare AS INTEGER
  floors AS INTEGER `(-ve for downwards)
  zord AS INTEGER
ENDTYPE

REM *****
REM ***      Global variables      ***
REM *****
GLOBAL DIM Lifts(27) AS ElevatorType
GLOBAL DIM board(100) AS INTEGER
GLOBAL playerat AS INTEGER

REM *****
REM ***      Constants      ***
REM *****
REM *** Object constants ***
#CONSTANT boardobj      1
#CONSTANT playerobj     2
#CONSTANT diceobj       3
#CONSTANT firstliftobj  4 `27 lifts IDs 4 - 30

REM *****
REM ***      Main game logic      ***
REM *****
SetUpGame()
REPEAT
  thrown = RollDice()
  MovePlayer(thrown)
UNTIL playerat = 100
EndScreen()
END

REM *****
REM ***      Level 1 routines      ***
REM *****
FUNCTION SetUpGame()
  InitialiseData()
  InitialiseVisuals()
ENDFUNCTION

FUNCTION RollDice()
ENDFUNCTION 1

FUNCTION MovePlayer(thrown)
ENDFUNCTION

FUNCTION EndScreen()
ENDFUNCTION

```

continued on next page

LISTING-42.2

(continued)

The Game So Far

```
REM *****
REM ***      Level 2 routines      ***
REM *****
FUNCTION InitialiseData()
    InitialiseLifts()
    InitialiseBoard()
    REM *** Seed random number generator ***
    RANDOMIZE TIMER()
    REM *** Start player at square 1 ***
    playerat = 1
ENDFUNCTION

FUNCTION InitialiseVisuals()
    SET DISPLAY MODE 1280,1024,32
    LoadBoard()
    LoadPlayerCharacter()
    LoadDice()
    PositionCameras()
    SET AMBIENT LIGHT 50
ENDFUNCTION

REM *****
REM ***      Level 3 routines      ***
REM *****
FUNCTION InitialiseLifts()
    REM *** Lifts start at square ***
    DATA 10,13,18,20,24,25,30,34,35
    DATA 39,40,49,50,53,54,59,60,65
    DATA 68,70,72,75,80,83,87,90,97
    REM *** Lifts end at square ***
    DATA 11,33,43,21,64,56,31,27,55
    DATA 19,41,12,51,48,67,79,61,76
    DATA 88,71,52,95,81,58,74,91,77
    REM *** Floors covered by lifts ***
    DATA 1,2,3,1,4,3,1,-1,2
    DATA -2,1,-3,1,-1,1,2,1,1
    DATA 2,1,-2,2,1,-3,-1,1,-2
    REM *** z-ordinate of lift's position ***
    DATA 47,23,-27,-47,-17,-7,47,13,3
    DATA -37,-47,33,47,23,13,-37,-47,-7
    DATA 23,47,33,3,-47,-27,13,47,-17
    REM *** Store lift IDs (firstliftobj to firstliftobj + 26)***
    FOR c = 1 TO 27
        lifts(c).ID = c + firstliftobj - 1
    NEXT c
    REM *** Store start square ***
    FOR c = 1 TO 27
        READ sq
        lifts(c).startsquare = sq
    NEXT c
    REM *** Store end square ***
    FOR c = 1 TO 27
        READ fin
        lifts(c).endsquare = fin
    NEXT c
    REM *** Store floors covered ***
    FOR c = 1 TO 27
        READ fl
        lifts(c).floors = fl
    NEXT c
    REM *** Store z-ord of lift ***
    FOR c = 1 TO 27
        READ z
        lifts(c).zord = z
    NEXT c
ENDFUNCTION
```

continued on next page

LISTING-42.2

(continued)

The Game So Far

```
FUNCTION InitialiseBoard()
  `Square empty (0) or lift (lift ID)
  DATA -1,-1,-1,-1,-1, -1,-1,-1,-1,1
  DATA -1,-1,2,-1,-1, -1,-1,3,-1,4
  DATA -1,-1,-1,5,6, -1,-1,-1,-1,7
  DATA -1,-1,-1,8,9, -1,-1,-1,10,11
  DATA -1,-1,-1,-1,-1, -1,-1,-1,12,13
  DATA -1,-1,14,15,-1, -1,-1,-1,16,17
  DATA -1,-1,-1,-1,18, -1,-1,19,-1,20
  DATA -1,21,-1,-1,22, -1,-1,-1,-1,23
  DATA -1,-1,24,-1,-1, -1,25,-1,-1,26
  DATA -1,-1,-1,-1,-1, -1,27,-1,-1,-1
  REM *** Store this data in board array ***
  FOR c = 1 TO 100
    READ board(c)
    IF board(c) = -1
      board(c) = 0
    ENDIF
  NEXT c
ENDFUNCTION

FUNCTION LoadBoard()
  REM *** Load, scale and position board ***
  LOAD OBJECT "board.x",1
  SCALE OBJECT 1, 1000,1000,1000
  YROTATE OBJECT 1, 90
  POSITION OBJECT 1,5,50,0
  REM *** Make black areas invisible ***
  REM *** These are the holes in the floor ***
  REM *** through which the lifts pass ***
  SET OBJECT TRANSPARENCY 1,1
  AddElevators()
ENDFUNCTION

FUNCTION LoadPlayerCharacter()
  REM *** Load, scale and position player's character ***
  LOAD OBJECT "H-android-move.x",playerobj
  SCALE OBJECT playerobj,150,150,150
  YROTATE OBJECT playerobj,180
  POSITION OBJECT playerobj,2,0,-45 `square 1
  REM *** Set the animation play speed ***
  SET OBJECT SPEED playerobj,15
ENDFUNCTION

FUNCTION LoadDice()
  REM *** Load, scale and position dice ***
  LOAD OBJECT "dice.dbo",diceobj
  POSITION OBJECT diceobj,30,15,45
  SCALE OBJECT diceobj,50,50,50
ENDFUNCTION

FUNCTION PositionCameras()
  AUTOCAM OFF
  REM *** Set up player's camera ***
  MAKE CAMERA 1
  SET CAMERA VIEW 1, 0,0,1279,1023
  POSITION CAMERA 1,3,5,-52
  POINT CAMERA 1,3,2,-40
  REM *** Set up dice's camera ***
  MAKE CAMERA 2
  POSITION CAMERA 2,50,43,0
  POINT CAMERA 2,30,15,45
  SET CAMERA FOV 2, 10
  SET CAMERA VIEW 2,1100,20,1220,120
ENDFUNCTION
```

continued on next page

LISTING-42.2

(continued)

The Game So Far

```
REM *****
REM ***           Level 4           ***
REM *****
FUNCTION AddElevators()
    REM *** Create the first lift ***
    LOAD OBJECT "lift.dbo",firstliftobj
    REM *** Create others as clones ***
    FOR c = 2 TO 27
        CLONE OBJECT lifts(c).ID,firstliftobj
    NEXT c
    REM *** Position all the lifts ***
    FOR c = 1 TO 27
        REM *** Rotate lift and scale to match floors ***
        liftID = lifts(c).ID
        YROTATE OBJECT liftID,-90
        SCALE OBJECT liftID,100,10*ABS(lifts(c).floors),100
        REM *** position up lifts at start square ***
        IF lifts(c).floors > 0
            POSITION OBJECT liftID,-4.5,
            ↵ (lifts(c).startsquare-1)/10*10,lifts(c).zord
        ELSE
            REM *** And down ones at the finish square ***
            POSITION OBJECT liftID,-4.5,
            ↵ (lifts(c).endsquare-1)/10*10,lifts(c).zord
            REM *** down lifts need platform at top ***
            OFFSET LIMB liftID,2,0,100,-2
        ENDIF
    NEXT c
ENDFUNCTION
```

Activity 42.7

Extend your program to match the code given in LISTING-42.2.

Adding *RollDice()*

With all the level 4, 3 and 2 functions now coded, we can return to the level 1 routines. The next of these to be coded is the *RollDice()* function which needs to show the dice being rolled (this appears in camera 2's output) and also returns the number shown on the dice.

Making what the dice shows on its top surface match the value returned by this function adds to the complexity of the code.

```
FUNCTION RollDice()
    REM *** Make dice spin ***
    FOR c = 0 TO 100
        PITCH OBJECT UP diceobj,15`RND(15)+1
        TURN OBJECT LEFT diceobj,15`RND(15)+1
        ROLL OBJECT LEFT diceobj,15 `RND(15)+1
        WAIT 1
    NEXT c
    REM *** Generate the value to be returned ***
    value = RND(5)+1
    REM *** make the dice show the value generated ***
    SELECT value
        CASE 1
            ROTATE OBJECT diceobj,-90,0,0
        ENDCASE
        CASE 2
            ROTATE OBJECT diceobj,180,0,0
        ENDCASE
        CASE 3
            ROTATE OBJECT diceobj,0,0,-90
```



```

ENDCASE
CASE 4
    ROTATE OBJECT diceobj,0,0,90
ENDCASE
CASE 5
    ROTATE OBJECT diceobj,0,0,0
ENDCASE
CASE 6
    ROTATE OBJECT diceobj,90,0,0
ENDCASE
ENDSELECT
WAIT KEY
ENDFUNCTION value

```

Activity 42.8

In your program, replace the *RollDice()* test stub with the code given above.

Adding *MovePlayer()*

Now we come to the main routine of the program. The task it has to do is quite simple in concept; move the player forward a number of squares corresponding to the value thrown on the dice.

Complications arise because of the elevators. If a character stops at an elevator square, then he must be moved to the elevator's destination. Also, if, while moving, the character arrives at the end of a row, an elevator must take the character to the next floor. During all of this, the main camera must follow the player's character.

This time, rather than give you the completed routine, we'll build it up step-by-step, making it more sophisticated as we go. We'll start by just trying to get the movement along the bottom row correct. A first attempt at the logic required might be:

thrown is a parameter of the *MovePlayer()* function and *playerat* is a global variable.

```

Start the character's animation (appears to be walking)
FOR thrown times DO
    Find the current coordinates of the character
    Move character one square
ENDFOR
Stop character animation and reset to frame 1
Add thrown to playerat

```

This can be coded as:

```

FUNCTION MovePlayer(thrown)
    REM *** Animate the character ***
    LOOP OBJECT playerobj
    REM *** Move character a square at a time ***
    FOR squares = 1 TO thrown
        REM *** Find out character's current position ***
        x# = OBJECT POSITION X(playerobj)
        y# = OBJECT POSITION Y(playerobj)
        z# = OBJECT POSITION Z(playerobj)
        POSITION OBJECT playerobj,x#,y#,z#+10
        WAIT 100
    NEXT squares
    REM *** Stop the animation ***
    STOP OBJECT playerobj
    SET OBJECT FRAME playerobj,1
    REM *** Update character's position ***
    playerat = playerat + thrown
ENDFUNCTION

```

Activity 42.9

Replace the *MovePlayer()* test stub with the code given. How good are the results?

The biggest problem is that the character does not move smoothly. To deal with this, we need to move the model using smaller steps.

If we change the code:

```
REM *** Move character a square at a time ***
FOR squares = 1 TO thrown
  REM *** Find out character's current position ***
  x# = OBJECT POSITION X(playerobj)
  y# = OBJECT POSITION Y(playerobj)
  z# = OBJECT POSITION Z(playerobj)
  POSITION OBJECT playerobj, x#, y#, z#+10
  WAIT 100
NEXT squares
```

to

```
REM *** Move character one square at a time ***
FOR squares = 1 TO thrown
  REM *** Find out character's current position ***
  x# = OBJECT POSITION X(playerobj)
  y# = OBJECT POSITION Y(playerobj)
  z# = OBJECT POSITION Z(playerobj)
  FOR move# = 1 TO 10 STEP 0.5
    POSITION OBJECT playerobj, x#, y#, z#+move#
    WAIT 100
  NEXT move#
NEXT squares
```

we can achieve a more realistic movement.

Activity 42.10

Make the changes described above and test your program again.

The next obvious problem is that the camera has been left behind when it should be following the character.

This is easily solved by moving the camera in unison with the character. The character was originally placed at position (2,0,-45) and camera 1 at (3,5,-52), so to keep the same relative positions, the camera needs to maintain a distance of (1,5,-7) from the character. This can be achieved using the line:

```
POSITION CAMERA 1, x#+1, y#+5, z#+move#-7
```

Activity 42.11

Add the above line at an appropriate point in your *MovePlayer()* routine and test the program.

Does the camera move with the character?

What happens to the character after a few moves?

The next feature we need to add to the *MovePlayer()* function is to detect when the character has reached the end of the corridor and then use the elevator to move to the next level. This means we need to add the following logic to the function:

```
IF at end of a level THEN
    Use elevator
ENDIF
```

However, we have to be careful. If we were currently in square 6 and then threw a 5, the character would move to square 11, taking the elevator up from square 10 as part of the move. However, if we started on square 6 and threw a 4, the character should move to square 10 and, although this is the base of an elevator, the move has already been completed, so the lift should not be used on this turn.

We can handle these two situations if we rework the logic of *MovePlayer()* slightly, changing it from:

```
Start the character's animation (appears to be walking)
FOR thrown times DO
    Find the current coordinates of the character
    Move character one square
ENDFOR
Stop character animation and reset to frame 1
Add thrown to playerat
```

to

```
Start the character's animation (appears to be walking)
FOR thrown times DO
    Find the current coordinates of the character
    IF at end of level THEN
        Use elevator
    ELSE
        Add 1 to playerat
    ENDIF
    Move character one square
ENDFOR
Stop character animation and reset to frame 1
```

Notice that *playerat* is now incremented a square at a time and only when the elevator is not being used (using the elevator will handle the updating of *playerat* separately). This lets us know where the character is at any time during its move. Notice also that we check for *end of level* before moving the character or incrementing *playerat*. If you think about it, you'll see that this handles both the situations of stopping at the end of a level without using the elevator and using the elevator on the way to a later square during a move.

If we implement checking for the *end of level* and using the elevator as two new functions, we can recode *MovePlayer()* as:

```
FUNCTION MovePlayer(thrown)
    REM *** Animate the character ***
    LOOP OBJECT playerobj
    REM *** Move character one square at a time ***
    FOR squares = 1 TO thrown
        REM *** Find out character's current position ***
        x# = OBJECT POSITION X(playerobj)
        y# = OBJECT POSITION Y(playerobj)
        z# = OBJECT POSITION Z(playerobj)
        IF AtEndOfLevel()
            UseElevator()
        ELSE
            INC playerat
```

```

    ENDIF
    FOR move# = 1 TO 10 STEP 0.5
        POSITION OBJECT playerobj,x#,y#,z#+move#
        POSITION CAMERA 1,x#+1, y#+5,move#-7
        WAIT 100
    NEXT move#
NEXT squares
REM *** Stop animation ***
STOP OBJECT playerobj
SET OBJECT FRAME playerobj,1
ENDFUNCTION

```

The function *AtEndOfLevel()* will return 1 if the character is at the end of a level (that is on square 10, 20, 30, etc.), otherwise zero will be returned.

Activity 42.12

Update your version of *MovePlayer()* to match the logic given.

Add a test stub for *UseElevator()*.

Code the *AtEndOfLevel()* function to implement the description given above.

Being at the end of a level is not the only time when an elevator is used. Should the character stop on some other square containing an elevator, that elevator must be taken, moving the character to a new position on the board dependent on the elevator's destination.

So, the *MovePlayer()* function should end with the logic:

```

    IF on square with elevator AND not at the end of a level THEN
        Use elevator
    ENDIF

```

Activity 42.13

What term can be used to determine if a square contains an elevator?

Convert the logic given above to DarkBASIC Pro code.

Update your version of *MovePlayer()* to incorporate this logic.

Adding *UseElevator()*

Using the elevator requires several distinct movements of the character. These are listed below

- Line the character up with the elevator. This may involve the character in moving forward slightly.
- Turn the character to face the elevator.
- Move the character onto the elevator platform
- Move the elevator and character to elevator's destination.
- Turn the character away from the elevator.
- Move the character to the centre of the new square.

- Face the character in the direction it is to travel.
- Return the elevator platform to its original position.

At all stages the camera needs to be tracking the character's movements. Since the elevator may be used part way through a move, it's probably best to halt the character's animation at the start of this function and resume it at the end.

The various turning manoeuvres can be handled by a single routine, but for the other stages, it would probably be best to create a function to implement each step. This means that *UseElevator()* can be coded as:

```

FUNCTION UseElevator()
    REM *** Stop character movement ***
    STOP OBJECT playerobj
    SET OBJECT FRAME playerobj,1
    MovePlayerToElevator()
    TurnPlayer(-90)
    MoveOntoPlatform()
    liftnum = MoveElevator()
    TurnPlayer(-180)
    MoveOffPlatform()
    TurnPlayer(90)
    ReturnElevator(liftnum)
    RepositionCamera()
    REM *** Resume character movement ***
    LOOP OBJECT playerobj
ENDFUNCTION

```

Notice that the lift number (*liftnum*) is returned by *MoveElevator()*. This value is later passed to *ReturnElevator()* to identify which elevator's platform must be returned to its starting position.

Activity 42.14

Replace your *UseElevator()* stub with the code given above and create the appropriate new stubs in the level 3 section of the program.

Note:

MoveElevator() returns the number of the lift being used.
TurnPlayer() takes the angle through which the player is to be turned.
ReturnElevator() takes the number of the lift being returned to its original position.

Adding *MovePlayerToElevator()*

We need to get the player's character level with the lift platform before turning the character to face the lift. We can get the exact position of the lift from the *lifts* array. The logic for this routine is:

```

Start the character's animation
Find the coordinates of the character's current position
FOR z = current z-ordinate of character TO lift's z-ordinate DO (steps of 0.2)
    Reposition character at (x,y,z)
    Wait 50 milliseconds
ENDFOR
Stop the character animation and reset to frame 1

```

Activity 42.15

Convert the logic given above to DarkBASIC Pro code and replace the function's stub with your code.

Check that the character moves correctly. (HINT: you may want to add a WAIT KEY after the function is called in *UseElevator()*. This will give you a chance to check out the character movement before the program continues.)

Adding *TurnPlayer()*

This routine takes a parameter specifying the angle through which the character is to be turned. For anti-clockwise movement, use a negative value.

The routine has the following logic:

```
IF the angle is negative THEN
    Set stepsize to -1
ELSE
    Set stepsize to 1
ENDIF
Set startangle as the character's current rotation about its y-axis
FOR degree = startangle TO startangle+angle DO (steps of stepsize)
    Rotate character to degree° about its y-axis
    Wait 1 millisecond
ENDFOR
```

Activity 42.16

Create the *TurnPlayer()* function using the logic given and test it out in your program. (HINT: you may want to move the WAIT KEY you added previously.)

Adding *MoveOntoPlatform()*

With the character facing in the correct direction, we now need to move it onto the elevator's platform. The logic required for this is:

```
Start the character's animation
Find the coordinates of the character's current position
FOR newx = x-ord TO x-ord - 5 DO (steps of -0.1))
    Reposition character at (x,y,z)
    Wait 50 millisecond
ENDFOR
Stop the character animation and reset to frame 1
```

Notice that the routine's logic is very similar to *MovePlayerToElevator()*.

Activity 42.17

Add a *MoveOntoPlatform()* function based on the logic given. Test the program and ensure that the character walks onto the platform.

Adding *MoveElevator()*

This routine moves the platform and the character standing on it to a new square. The routine must be general enough so that it will work for elevators of different

lengths and for ones which travel down as well as up.

Since the elevator platform is a limb, its movement will be specified as an offset from its original position within the model. The model has the platform at the bottom for lifts that travel upwards (that is, it starts with an offset of zero along the y-axis). The platform will be at the top when the offset is changed to 100 (the models have an original height of 100 units). Downward travelling elevators start with an offset of 100 and this changes to zero to make the platform descend.

The function requires the following logic:

```
Get the lift's position in the lifts array from the board array
Get the lift's ID from the lifts array>
Calculate the height of the lift as 10 * the number of floors it covers
                                                                    (neg value for lifts which travel down)
IF the height is positive THEN
    Set liftdirection to 1
ELSE
    Set liftdirection to -1
ENDIF
Find the coordinates of the character's current position
IF the lift is going up THEN
    Set liftendoffset to 100
ELSE
    Set liftendoffset to 0
ENDIF
Set liftstartoffset to 100 - liftendoffset
Set unitsperpercent to height/100

FOR move = liftstartoffset TO liftendoffset DO (steps of liftdirection)
    Offset platform limb (0,move,-2)
    Position character at (x,y+unitsperperc*move,z)
    Wait 10 milliseconds
ENDFOR
Set playerat to the end square of the elevator
```

Activity 42.18

Add a *MoveElevator()* function based on the logic given. Test the program and ensure that the platform moves correctly.

Adding *MoveOffPlatform()*

To move the character off the platform, after having been made to face in the appropriate direction, it is just of matter of walking the character forward. This routine needs the following logic:

```
Start the character's animation
Find the coordinates of the character's current position
FOR newx = x-ord TO x-ord +5 DO (steps of 0.1)
    Reposition character at (x,y,z)
    Wait 50 millisecond
ENDFOR
Stop the character animation and reset to frame 1
```

Activity 42.19

Add a *MoveOffPlatform()* function based on the logic given. Test the program and ensure that the character walks off the platform.

Adding *ReturnElevator()*

This routine moves the elevator's platform back to its original position. The lift's position within the *lifts* array needs to be passed to the function since this information can no longer be obtained from the *board* array. The routine requires the following logic:

```
Get the lift's ID
Calculate the height of the lift (negative for a lift that has moved down)
IF the height is positive THEN
    Set liftdirection to -1
ELSE
    Set liftdirection to 1
ENDIF
IF liftdirection = -1 THEN
    Set liftendoffset to 0
ELSE
    Set liftendoffset to 100
ENDIF
Set liftstartoffset to 100 - liftendoffset
Set unitsperperc to height / 100.0
FOR move = liftstartoffset TO liftendoffset (step by liftdirection)
    Offset platform limb (2,0,move,-2)
    WAIT 10 milliseconds
ENDFOR
```

Activity 42.20

Add a *ReturnElevator()* function based on the logic given. Test the program and ensure that the platform returns to its original position.

Adding *RepositionCamera()*

With the character on a new level, we need to move the camera behind it, looking down the new corridor.

The routine should implement the following logic:

```
Get the character's coordinates
Move the camera behind the character
Point the camera down the new corridor.
```

This requires the following code:

```
FUNCTION RepositionCamera()
    REM *** Get position of character ***
    x# = OBJECT POSITION X(playerobj)
    # = OBJECT POSITION Y(playerobj)
    # = OBJECT POSITION Z(playerobj)
    REM *** Position and point camera ***
    POSITION CAMERA 1, x#+1,y#+5,z#+7
    POINT CAMERA 1,x#+1,y#+2,z-5
ENDFUNCTION
```

Activity 42.21

Add the routine given above and test out the program.

What problem occurs?

Fixing the Shortcomings

We've been so busy writing code to work on the first corridor, that we haven't taken into account that the second corridor travels in the opposite direction.

We can either write a whole new set of routines to handle movement in this direction, or we can make the previous routines more general so that they can handle travel in both directions. This second method is the best approach since the changes required are actually quite small.

We'll need a routine that can give us an indication of which direction the character is travelling in:

```
FUNCTION FindDirectionOfTravel()  
    REM *** IF odd floor, 1 else -1 ***  
    IF ((playerat-1)/10) mod 2 = 0  
        result = 1  
    ELSE  
        result = -1  
    ENDIF  
ENDFUNCTION result
```

This function returns 1 if the character is on an odd floor and -1 if on an even floor.

Activity 42.22

Add the above function to your program as a Level 4 routine.

Fixing *RepositionCamera()*

As the player's character travels along a corridor it is moving along the z-axis. On odd corridors (1, 3, 5, etc.) it is moving in a positive direction (its z-ordinate becoming greater); on even corridors it is travelling in a negative direction. So, if the camera is to be behind the character it must have a lower z-ordinate than the character when travelling along odd corridors and a higher z-ordinate when moving along even corridors (see FIG-42.4).

FIG-42.4 The Camera's Position and Direction of Travel



In fact, the camera's z-ordinate should be 7 units less than the character's when travelling along odd corridors and 7 units more when travelling along even corridors. The direction in which the camera is pointing should be +5 z-units for odd corridors and -5 z-units for even corridors.

So, to reposition the camera when we reach a new level, we need to take into account what level we are on. This requires the following changes to the

RepositionCamera() function:

```
FUNCTION RepositionCamera()  
  REM *** Get position of character ***  
  x# = OBJECT POSITION X(playerobj)  
  y# = OBJECT POSITION Y(playerobj)  
  z# = OBJECT POSITION Z(playerobj)  
  REM *** IF odd corridor, camera -7 z-units ***  
  IF FindDirectionOfTravel() = 1  
    POSITION CAMERA 1, x#+1,y#+5,z#-7  
    POINT CAMERA 1,x#+1,y#+2,z#+5  
  ELSE  
    REM *** Even corridor, +7 z-units ***  
    POSITION CAMERA 1, x#+1,y#+5,z#+7  
    POINT CAMERA 1, x#+1,y#+2,z#-5  
  ENDIF  
ENDFUNCTION
```

Notice that the only difference between the two options is the sign of the 7 and 5 values when calculating the z-ordinates. Since *FindDirectionOfTravel()* returns 1 or -1 we could rewrite the function without the IF statement as:

```
FUNCTION RepositionCamera()  
  REM *** Get position of character ***  
  x# = OBJECT POSITION X(playerobj)  
  y# = OBJECT POSITION Y(playerobj)  
  z# = OBJECT POSITION Z(playerobj)  
  REM *** Position camera ***  
  POSITION CAMERA 1, x#+1,y#+5,z# -(7*FindDirectionOfTravel())  
  POINT CAMERA 1, x#+1,y#+2,z# + (5*FindDirectionOfTravel())  
ENDFUNCTION
```

Activity 42.23

Modify your *RepositionCamera()* function as described above.

Fixing *MovePlayer()*

The camera and character are both moved in the *MovePlayer()* function, so we also need to take the direction of travel into account in that routine.

```
FUNCTION MovePlayer(thrown)  
  REM *** Animate the character ***  
  LOOP OBJECT playerobj  
  REM *** Move character one square at a time ***  
  FOR squares = 1 TO thrown  
    IF AtEndOfLevel()  
      REM *** Move to next level ***  
      UseElevator()  
    ELSE  
      REM *** Update character's position ***  
      INC playerat  
      REM *** Find out character's current position ***  
      x# = OBJECT POSITION X(playerobj)  
      y# = OBJECT POSITION Y(playerobj)  
      z# = OBJECT POSITION Z(playerobj)  
      REM *** Find direction of travel ***  
      direction = FindDirectionOfTravel()  
      FOR move# = 1 TO 10 STEP 0.5  
        POSITION OBJECT playerobj,x#,y#,  
          ↵ z#+move#*direction  
        POSITION CAMERA 1,x#+1, y#+5,  
          ↵ z#+(move#-7*direction)
```

```

        WAIT 100
    NEXT move#
ENDIF
NEXT squares
REM *** IF stopped at elevator, use it ***
IF board(playerat) <> 0 AND AtEndOfLevel() = 0
    UseElevator()
ENDIF
REM *** Stop animation ***
STOP OBJECT playerobj
SET OBJECT FRAME playerobj,1
ENDFUNCTION

```

Activity 42.24

Update your *MovePlayer()* routine to match the code given.

Fixing *UseElevator()*

When the character uses an elevator, the angle of turn depends on which direction the character is facing, so we need to modify the parameter of the calls to *TurnPlayer()* in the *UseElevator()* function. Both calls should be re-coded as:

```
TurnPlayer(-90*FindDirectionOfTravel())
```

Activity 42.25

Update your *UseElevator()* routine to match the changes given above.

Fixing *MovePlayerToElevator()*

How the player moves in line with the elevator's platform is also dependent on the direction in which the character is travelling. In *MovePlayerToElevator()* the line

```
FOR z# = OBJECT POSITION Z(playerobj) TO
  ↪ lifts(board(playerat)).zord STEP 0.2
```

must be changed to

```
FOR z# = OBJECT POSITION Z(playerobj) TO
  ↪ lifts(board(playerat)).zord STEP 0.2*FindDirectionOfTravel()
```

Activity 42.26

Update your *MovePlayerToElevator()* routine to match the change given above.

Fixing *MoveElevator()*

The camera moves in synchronisation with the elevator, but which side of the character the camera is on depends on the floor on which the elevator starts its journey, so *MoveElevator()* must have the line

```
POSITION CAMERA 1, playerx#,OBJECT POSITION Y(playerobj)+5,
  ↪ playerz#-7
```

changed to

```
POSITION CAMERA 1, playerx#,OBJECT POSITION Y(playerobj)+5,  
↵playerz#-(7*FindDirectionOfTravel())
```

Activity 42.27

Change your *MoveElevator()* function as described.

Try out the game.

Adding *EndGame()*

Only one routine left to code! The *EndGame()* function will delete all objects on the screen and then show a Rotating 3D version of the text "GAME OVER" for 5 seconds before the program terminates.

The routine requires the following code:

```
FUNCTION EndScreen()  
  REM *** Delete all existing objects ***  
  DELETE OBJECTS 1,30  
  REM *** Load and position text model ***  
  LOAD OBJECT "endtext.x",1  
  POSITION OBJECT 1, 0,0,100  
  REM *** Rotate text for 5 seconds ***  
  now = TIMER()  
  WHILE TIMER() - now < 5000  
    PITCH OBJECT UP 1,1  
  ENDWHILE  
ENDFUNCTION
```

Activity 42.28

Add the *EndGame()* code to your program and test the completed game.

Game Review

We've created the most complex code so far - but have we created a game?

There's certainly not a great deal of player interaction in this version of the game and there are no skills involved in achieving the game's goal, but we could do things to improve it.

For a start we could have multiple players and then there is at least someone to compete against.

Rather than use the dice to decide the number of squares moved, we could ask the player up to six questions. Movement would then be based on the number of correct questions.

We could add some animated models on some of the squares. These need not have any purpose but would add to the overall eye candy effect.

Solutions

Activity 42.1

```
REM *****
REM ***      Structures      ***
REM *****
REM *** Elevator details ***
TYPE ElevatorType
  ID AS INTEGER
  startsquare AS INTEGER
  endsquare AS INTEGER
  floors AS INTEGER `(-ve for downwards)
  zord AS INTEGER
ENDTYPE
REM *****
REM ***      Global variables      ***
REM *****
GLOBAL DIM Lifts(27) AS ElevatorType
GLOBAL DIM board(100) AS INTEGER
GLOBAL playerat AS INTEGER
REM *****
REM ***      Constants      ***
REM *****
REM *** Object constants ***
#CONSTANT boardobj      1
#CONSTANT playerobj     2
#CONSTANT diceobj       3
#CONSTANT firstliftobj  4 `IDs 4-30

REM *****
REM ***      Main game logic      ***
REM *****
SetUpGame()
REPEAT
  thrown = RollDice()
  MovePlayer(thrown)
UNTIL playerat = 100
EndScreen()
END

REM *****
REM ***      Level 1 routines      ***
REM *****
FUNCTION SetUpGame()
ENDFUNCTION

FUNCTION RollDice()
ENDFUNCTION 1

FUNCTION MovePlayer(thrown)
ENDFUNCTION

FUNCTION EndScreen()
ENDFUNCTION

REM *****
REM ***      Level 2 routines      ***
REM *****
FUNCTION InitialiseData()
  InitialiseLifts()
  InitialiseBoard()
  REM *** Start player at square 1 ***
  playerat = 1
  REM *** Seed random number generator ***
  RANDOMIZE TIMER()
ENDFUNCTION

FUNCTION InitialiseVisuals()
ENDFUNCTION

REM *****
REM ***      Level 3 routines      ***
REM *****
FUNCTION InitialiseLifts()
ENDFUNCTION

FUNCTION InitialiseBoard()
ENDFUNCTION
```

Activity 42.2

No solution required.

Activity 42.3

```
REM *****
REM ***      Structures      ***
REM *****
REM *** Elevator details ***
TYPE ElevatorType
  ID AS INTEGER
  startsquare AS INTEGER
  endsquare AS INTEGER
  floors AS INTEGER `(-ve for downwards)
  zord AS INTEGER
ENDTYPE
```

Activity 42.4

Level 3 should now contain the following code:

```
REM *****
REM ***      Level 3 routines      ***
REM *****
```

```

FUNCTION InitialiseLifts()
  REM *** Lifts start at
  DATA 10,13,18,20,24,25,30,34,35
  DATA 39,40,49,50,53,54,59,60,65
  DATA 68,70,72,75,80,83,87,90,97
  `Lifts end at
  DATA 11,33,43,21,64,56,31,27,55
  DATA 19,41,12,51,48,67,79,61,76
  DATA 88,71,52,95,81,58,74,91,77
  `Floors covered
  DATA 1,2,3,1,4,3,1,-1,2
  DATA -2,1,-3,1,-1,2,1,1
  DATA 2,1,-2,2,1,-3,-1,1,-2
  `Xord
  DATA 47,23,-27,-47,-17,-7,47,13,3
  DATA -37,-47,33,47,23,13,-37,-47,-7
  DATA 23,47,33,3,-47,-27,13,47,-17
  REM *** Store lift IDs (31-57)***
  FOR c = 1 TO 27
    lifts(c).ID = c + firstliftobj -1
  NEXT c
  REM *** Store start square ***
  FOR c = 1 TO 27
    READ sq
    lifts(c).startsquare = sq
  NEXT c
  REM *** Store end square ***
  FOR c = 1 TO 27
    READ fin
    lifts(c).endsquare = fin
  NEXT c
  REM *** Store floors covered ***
  FOR c = 1 TO 27
    READ fl
    lifts(c).floors = fl
  NEXT c
  REM *** Store x-ord of lift ***
  FOR c = 1 TO 27
    READ x
    lifts(c).zord = x
  NEXT c
ENDFUNCTION

FUNCTION InitialiseBoard()
  `Square empty (0) or lift (lift ID)
  DATA -1,-1,-1,-1,-1, -1,-1,-1,-1,1
  DATA -1,-1,2,-1,-1, -1,-1,3,-1,4
  DATA -1,-1,-1,5,6, -1,-1,-1,-1,7
  DATA -1,-1,-1,8,9, -1,-1,-1,10,11
  DATA -1,-1,-1,-1,-1, -1,-1,-1,12,13
  DATA -1,-1,14,15,-1, -1,-1,-1,16,17
  DATA -1,-1,-1,-1,18, -1,-1,19,-1,20
  DATA -1,21,-1,-1,22, -1,-1,-1,-1,23
  DATA -1,-1,24,-1,-1, -1,25,-1,-1,26
  DATA -1,-1,-1,-1,-1, -1,27,-1,-1,-1
  REM *** Store data in board array ***
  FOR c = 1 TO 100
    READ board(c)
    IF board(c) = -1
      board(c) = 0
    ENDIF
  NEXT c
ENDFUNCTION

```

Activity 42.5

In level 2, the code

```

FUNCTION InitialiseVisuals()
ENDFUNCTION

```

should have been replaced by

```

FUNCTION InitialiseVisuals()
  REM *** Set screen resolution ***
  SET DISPLAY MODE 1280,1024,32

```

```

LoadBoard()
LoadPlayerCharacter()
LoadDice()
PositionCameras()
REM *** Increase ambient light ***
SET AMBIENT LIGHT 50
ENDFUNCTION

```

The following code should have been added to level 3

```

FUNCTION LoadBoard()
ENDFUNCTION

FUNCTION LoadPlayerCharacter()
ENDFUNCTION

FUNCTION LoadDice()
ENDFUNCTION

FUNCTION PositionCameras()
ENDFUNCTION

FUNCTION SetLights()
ENDFUNCTION

```

Activity 42.6

No solution required.

Activity 42.7

No solution required.

Activity 42.8

The Level 1 code should read as follows:

```

REM *****
REM ***      Level 1 routines      ***
REM *****
FUNCTION SetUpGame()
  InitialiseData()
  InitialiseVisuals()
ENDFUNCTION

FUNCTION RollDice()
  REM *** Make dice spin ***
  FOR c = 0 TO 100
    PITCH OBJECT UP diceobj,15`RND(15)+1
    TURN OBJECT LEFT diceobj,15`RND(15)+1
    ROLL OBJECT LEFT diceobj,15`RND(15)+1
    WAIT 1
  NEXT c
  REM *** Generate return value ***
  value = RND(5)+1
  REM *** Make dice show value ***
  SELECT value
    CASE 1
      ROTATE OBJECT diceobj,-90,0,0
    ENDCASE
    CASE 2
      ROTATE OBJECT diceobj,180,0,0
    ENDCASE
    CASE 3
      ROTATE OBJECT diceobj,0,0,-90
    ENDCASE
    CASE 4
      ROTATE OBJECT diceobj,0,0,90
    ENDCASE
    CASE 5
      ROTATE OBJECT diceobj,0,0,0
    ENDCASE
    CASE 6

```

```

        ROTATE OBJECT diceobj,90,0,0
    ENDCASE
ENDSELECT
WAIT KEY
ENDFUNCTION value

FUNCTION MovePlayer(playerID, move)
ENDFUNCTION

FUNCTION EndScreen()
ENDFUNCTION

```

Activity 42.9

The character moves too quickly and the camera is left behind.

Activity 42.10

The code for *MovePlayer()* should now be:

```

FUNCTION MovePlayer(thrown)
    REM *** Find char's current position ***
    x# = OBJECT POSITION X(playerobj)
    y# = OBJECT POSITION Y(playerobj)
    z# = OBJECT POSITION Z(playerobj)
    REM *** Animate the character ***
    LOOP OBJECT playerobj
    REM *** Move char 1 square at a time ***
    FOR squares = 1 TO thrown
        FOR move# = 1 TO 10 STEP 0.5
            POSITION OBJECT playerobj,x#,y#,
            ↵ z#+ (squares-1)*10+move#
            WAIT 100
        NEXT move#
    NEXT squares
    REM *** Stop animation ***
    STOP OBJECT playerobj
    SET OBJECT FRAME playerobj,1
    REM *** Update character's position ***
    playerat = playerat + thrown
ENDFUNCTION

```

Activity 42.11

The camera now follows the character as it moves.

After a few moves the character moves off the board!

Activity 42.12

The following code should be added at the end of the Level 2 section.

```

FUNCTION AtEndOfLevel()
    IF playerat mod 10 = 0
        result = 1
    ELSE
        result = 0
    ENDIF
ENDFUNCTION result

FUNCTION UseElevator()
ENDFUNCTION

```

Activity 42.13

The expression required is:

```
board(playerat) <> 0
```

The DarkBASIC Pro equivalent is

```

IF (board(playerat) <> 0) AND
↵ (NOT AtEndOfLevel())
    UseElevator()
ENDIF

```

Notice that extra parentheses are required to conform to the language's syntax.

The new version of *MovePlayer()* is:

```

FUNCTION MovePlayer(thrown)
    REM *** Find out character's current
    position ***
    x# = OBJECT POSITION X(playerobj)
    y# = OBJECT POSITION Y(playerobj)
    z# = OBJECT POSITION Z(playerobj)
    REM *** Animate the character ***
    LOOP OBJECT playerobj
    REM *** Move char 1 square at a time ***
    FOR squares = 1 TO thrown
        IF AtEndOfLevel()
            UseElevator()
        ENDIF
        FOR move# = 1 TO 10 STEP 0.5
            POSITION OBJECT playerobj,x#,y#,
            ↵ z#+ (squares-1)*10+move#
            POSITION CAMERA 1,x#+1, y#+5,
            ↵ z#+ (squares-1)*10+move#-7
            WAIT 100
        NEXT move#
        REM *** Update char's position ***
        INC playerat
    NEXT squares
    REM *** Stop animation ***
    STOP OBJECT playerobj
    SET OBJECT FRAME playerobj,1
    IF (board(playerat) <> 0) AND
    ↵ (NOT AtEndOfLevel())
        UseElevator()
    ENDIF
ENDFUNCTION

```

Activity 42.14

The new test stubs in level 3 should be:

```

FUNCTION MovePlayerToLift()
ENDFUNCTION

FUNCTION TurnPlayer(angle)
ENDFUNCTION

FUNCTION MoveOntoPlatform()
ENDFUNCTION

FUNCTION MoveElevator()
ENDFUNCTION 1

FUNCTION MoveOffPlatform()
ENDFUNCTION

FUNCTION RepositionCamera()
ENDFUNCTION

FUNCTION ReturnElevator(liftnum)
ENDFUNCTION

```

Activity 42.15

The code for *MovePlayerToElevator()* is:

```
FUNCTION MovePlayerToLift()
```

```

REM *** Move character to lift ***
LOOP OBJECT playerobj
x# = OBJECT POSITION X(playerobj)
y# = OBJECT POSITION Y(playerobj)
FOR z# = OBJECT POSITION Z(playerobj) TO
↳lifts(board(playerat)).zord STEP 0.2
  POSITION OBJECT playerobj, x#,y#,z#
  WAIT 50
NEXT z#
STOP OBJECT playerobj
SET OBJECT FRAME playerobj,1
ENDFUNCTION

```

```

POSITION OBJECT playerobj,playerx#,
↳playery#+unitsperperc#*move,
↳playerz#
POSITION CAMERA 1, playerx#,
↳playery#+unitsperperc#*move+5,
↳playerz#-7
WAIT 10
NEXT move
playerat = lifts(liftnum).endsquare
ENDFUNCTION liftnum

```

Activity 42.19

The routine's code is:

```

FUNCTION MoveOffPlatform()
  LOOP OBJECT playerobj
  x# = OBJECT POSITION X(playerobj)
  y# = OBJECT POSITION Y(playerobj)
  z# = OBJECT POSITION Z(playerobj)
  FOR newx# = x# TO x#+5 STEP 0.1
    POSITION OBJECT playerobj, newx#,y#,z#
    WAIT 50
  NEXT moves
  STOP OBJECT playerobj
  SET OBJECT FRAME playerobj,1
ENDFUNCTION

```

Activity 42.20

The routine's code is:

```

FUNCTION ReturnElevator(liftnum)
  liftID = lifts(liftnum).ID
  height = lifts(liftnum).floors*10
  IF height > 0
    liftdirection = -1
  ELSE
    liftdirection = 1
  ENDIF
  IF liftdirection = -1
    liftendoffset = 0
  ELSE
    liftendoffset = 100
  ENDIF
  liftstartoffset = 100 - liftendoffset
  unitsperperc# = height / 100.0
  FOR move = liftstartoffset TO
↳liftendoffset STEP liftdirection
    OFFSET LIMB liftID,2,0,move,-2
    WAIT 10
  NEXT move
ENDFUNCTION

```

Activity 42.21

The function should be added at the end of the Level 3 code.

When the character moves to the next level, it is facing in the wrong direction. The camera is also incorrectly positioned.

Activity 42.22

No solution required.

Activity 42.23

No solution required.

Activity 42.16

The code for the function is:

```

FUNCTION TurnPlayer(angle)
  IF angle < 0
    stepsize = -1
  ELSE
    stepsize = 1
  ENDIF
  startangle = OBJECT ANGLE Y(playerobj)
  FOR degree = startangle TO
↳startangle + angle STEP stepsize
    YROTATE OBJECT playerobj, degree
    WAIT 1
  NEXT angle
ENDFUNCTION

```

Activity 42.17

The code for the function is:

```

FUNCTION MoveOntoPlatform()
  LOOP OBJECT playerobj
  x# = OBJECT POSITION X(playerobj)
  y# = OBJECT POSITION Y(playerobj)
  z# = OBJECT POSITION Z(playerobj)
  FOR newx# = x# TO x#-5 STEP -0.1
    POSITION OBJECT playerobj, newx#,y#,z#
    WAIT 50
  NEXT moves
  STOP OBJECT playerobj
  SET OBJECT FRAME playerobj,1
ENDFUNCTION

```

Activity 42.18

The code for the function is:

```

FUNCTION MoveElevator()
  liftnum = board(playerat)
  liftID = lifts(liftnum).ID
  height = lifts(liftnum).floors*10
  IF height > 0
    liftdirection = 1
  ELSE
    liftdirection = -1
  ENDIF
  playerx# = OBJECT POSITION X(playerobj)
  playery# = OBJECT POSITION Y(playerobj)
  playerz# = OBJECT POSITION Z(playerobj)
  IF liftdirection = 1
    liftendoffset = 100
  ELSE
    liftendoffset = 0
  ENDIF
  liftstartoffset = 100 - liftendoffset
  unitsperperc# = height / 100.0
  FOR move = liftstartoffset TO
↳liftendoffset STEP liftdirection
    OFFSET LIMB liftID,2,0,move,-2

```


Activity 42.24

No solution required.

Activity 42.25

No solution required.

Activity 42.26

No solution required.

Activity 42.27

No solution required.

Activity 42.28

The final version of the program is:

```
REM *****
REM *** Structures ***
REM *****
REM *** Elevator details ***
TYPE ElevatorType
  ID AS INTEGER
  startsquare AS INTEGER
  endsquare AS INTEGER
  floors AS INTEGER `(-ve for downwards)
  zord AS INTEGER
ENDTYPE
REM *****
REM *** Global variables ***
REM *****
GLOBAL DIM Lifts(27) AS ElevatorType
GLOBAL DIM board(100) AS INTEGER
GLOBAL playerat AS INTEGER
REM *****
REM *** Constants ***
REM *****
REM *** Object constants ***
#CONSTANT boardobj 1
#CONSTANT playerobj 2
#CONSTANT diceobj 3
#CONSTANT firstliftobj 4
`27 lifts IDs 4 - 30
REM *****
REM *** Main game logic ***
REM *****
SetUpGame()
REPEAT
  thrown = RollDice()
  MovePlayer(thrown)
UNTIL playerat > 2
EndScreen()
END

FUNCTION MakeCamera3()
  MAKE CAMERA 3
  POSITION CAMERA 3,
  ↵ OBJECT POSITION X(playerobj)+10,
  ↵ OBJECT POSITION Y(playerobj)+5,
  ↵ OBJECT POSITION Z(playerobj)
  POINT CAMERA 3,
  ↵ OBJECT POSITION X(playerobj),
  ↵ OBJECT POSITION Y(playerobj),
  ↵ OBJECT POSITION Z(playerobj)
  WAIT KEY
ENDFUNCTION
REM *****
REM *** Level 1 routines ***
REM *****
FUNCTION SetUpGame()
```

```
  InitialiseData()
  InitialiseVisuals()
ENDFUNCTION

FUNCTION RollDice()
  REM *** Make dice spin ***
  FOR c = 0 TO 100
    PITCH OBJECT UP diceobj,15`RND(15)+1
    TURN OBJECT LEFT diceobj,15`RND(15)+1
    ROLL OBJECT LEFT diceobj,15 `RND(15)+1
    WAIT 1
  NEXT c
  REM *** Generate value to be returned ***
  value = RND(5)+1
  REM *** make dice show its value ***
  SELECT value
    CASE 1
      ROTATE OBJECT diceobj,-90,0,0
    ENDCASE
    CASE 2
      ROTATE OBJECT diceobj,180,0,0
    ENDCASE
    CASE 3
      ROTATE OBJECT diceobj,0,0,-90
    ENDCASE
    CASE 4
      ROTATE OBJECT diceobj,0,0,90
    ENDCASE
    CASE 5
      ROTATE OBJECT diceobj,0,0,0
    ENDCASE
    CASE 6
      ROTATE OBJECT diceobj,90,0,0
    ENDCASE
  ENDSELECT
  WAIT KEY
ENDFUNCTION value

FUNCTION MovePlayer(thrown)
  REM *** Animate the character ***
  LOOP OBJECT playerobj
  REM *** Move char 1 square at a time ***
  FOR squares = 1 TO thrown
    IF AtEndOfLevel()
      REM *** Move to next level ***
      UseElevator()
    ELSE
      REM *** Update char's position ***
      INC playerat
      REM *** Find char's position ***
      x# = OBJECT POSITION X(playerobj)
      y# = OBJECT POSITION Y(playerobj)
      z# = OBJECT POSITION Z(playerobj)
      direction = FindDirectionOfTravel()
      FOR move# = 1 TO 10 STEP 0.5
        POSITION OBJECT playerobj,x#,y#,
        ↵ z#+move#*direction
        POSITION CAMERA 1,x#+1,y#+5,
        ↵ z#+(move#-7)*direction
        WAIT 100
      NEXT move#
    ENDIF
  NEXT squares
  REM *** IF stopped at lift, use it ***
  IF board(playerat) <> 0 AND
  ↵ AtEndOfLevel() = 0
    UseElevator()
  ENDIF
  REM *** Stop animation ***
  STOP OBJECT playerobj
  SET OBJECT FRAME playerobj,1
ENDFUNCTION

FUNCTION EndScreen()
  DELETE OBJECTS 1,30
  LOAD OBJECT "endtext.x",1
  POSITION OBJECT 1, 0,0,100
  now = TIMER()
```

```

WHILE TIMER() - now < 5000
    PITCH OBJECT UP 1,1
ENDWHILE
ENDFUNCTION

REM *****
REM *** Level 2 routines ***
REM *****
FUNCTION InitialiseData()
    InitialiseLifts()
    InitialiseBoard()
    REM *** Start player at square 1 ***
    playerat = 1
    REM *** Seed random number generator ***
    RANDOMIZE TIMER()
ENDFUNCTION

FUNCTION InitialiseVisuals()
    REM *** Set the screen resolution ***
    SET DISPLAY MODE 1280,1024,32
    LoadBoard()
    LoadPlayerCharacter()
    LoadDice()
    PositionCameras()
    REM *** Increase the ambient light ***
    SET AMBIENT LIGHT 50
ENDFUNCTION

FUNCTION AtEndOfLevel()
    IF playerat mod 10 = 0
        result = 1
    ELSE
        result = 0
    ENDIF
ENDFUNCTION result

FUNCTION UseElevator()
    REM *** Stop character movement ***
    STOP OBJECT playerobj
    SET OBJECT FRAME playerobj,1
    MovePlayerToElevator()
    TurnPlayer(-90*FindDirectionOfTravel())
    MoveOntoPlatform()
    liftnum = MoveElevator()
    TurnPlayer(-180)
    MoveOffPlatform()
    TurnPlayer(-90*FindDirectionOfTravel())
    RepositionCamera()
    ReturnElevator(liftnum)
    REM *** Resume character movement ***
    LOOP OBJECT playerobj
ENDFUNCTION

REM *****
REM *** Level 3 routines ***
REM *****
FUNCTION InitialiseLifts()
    REM *** Lifts start at square ***
    DATA 10,13,18,20,24,25,30,34,35
    DATA 39,40,49,50,53,54,59,60,65
    DATA 68,70,72,75,80,83,87,90,97
    REM *** Lifts end at square ***
    DATA 11,33,43,21,64,56,31,27,55
    DATA 19,41,12,51,48,67,79,61,76
    DATA 88,71,52,95,81,58,74,91,77
    REM *** Floors covered by lift ***
    DATA 1,2,3,1,4,3,1,-1,2
    DATA -2,1,-3,1,-1,1,2,1,1
    DATA 2,1,-2,2,1,-3,-1,1,-2
    REM *** z-ord of lifts' position ***
    DATA 47,23,-27,-47,-17,-7,47,13,3
    DATA -37,-47,33,47,23,13,-37,-47,-7
    DATA 23,47,33,3,-47,-27,13,47,-17
    REM *** Store lift IDs
    ↵(firstliftobj to firstliftobj + 26)***
    FOR c = 1 TO 27
        lifts(c).ID = c + firstliftobj - 1
    NEXT c

```

```

REM *** Store start square for lift ***
FOR c = 1 TO 27
    READ sq
    lifts(c).startsquare = sq
NEXT c
REM *** Store end square for lift ***
FOR c = 1 TO 27
    READ fin
    lifts(c).endsquare = fin
NEXT c
REM *** Store floors covered ***
FOR c = 1 TO 27
    READ fl
    lifts(c).floors = fl
NEXT c
REM *** Store z-ord of lift ***
FOR c = 1 TO 27
    READ z
    lifts(c).zord = z
NEXT c
ENDFUNCTION

FUNCTION InitialiseBoard()
    REM ***Square empty (0) or
    ↵levator's location in lifts array ***
    DATA -1,-1,-1,-1,-1, -1,-1,-1,-1,1
    DATA -1,-1,2,-1,-1, -1,-1,3,-1,4
    DATA -1,-1,-1,5,6, -1,-1,-1,-1,7
    DATA -1,-1,-1,8,9, -1,-1,-1,10,11
    DATA -1,-1,-1,-1,-1, -1,-1,-1,12,13
    DATA -1,-1,14,15,-1, -1,-1,-1,16,17
    DATA -1,-1,-1,-1,18, -1,-1,19,-1,20
    DATA -1,21,-1,-1,22, -1,-1,-1,-1,23
    DATA -1,-1,24,-1,-1, -1,25,-1,-1,26
    DATA -1,-1,-1,-1,-1, -1,27,-1,-1,-1
    REM *** Store this data in board ***
    FOR c = 1 TO 100
        READ board(c)
        IF board(c) = -1
            board(c) = 0
        ENDIF
    NEXT c
ENDFUNCTION

FUNCTION LoadBoard()
    REM *** Load, scale and model board ***
    LOAD OBJECT "board.x",1
    SCALE OBJECT 1, 1000,1000,1000
    YROTATE OBJECT 1, 90
    POSITION OBJECT 1,5,50,0
    REM *** Make black areas invisible ***
    REM *** These are holes in the floor ***
    REM *** through which the lifts pass ***
    SET OBJECT TRANSPARENCY 1,1
    AddElevators()
ENDFUNCTION

FUNCTION LoadPlayerCharacter()
    REM *** Load, scale and place char ***
    LOAD OBJECT "H-android-move.x",playerobj
    SCALE OBJECT playerobj,150,150,150
    YROTATE OBJECT playerobj,180
    POSITION OBJECT playerobj,2,0,-45
    REM *** Set the animation play speed ***
    SET OBJECT SPEED playerobj,15
ENDFUNCTION

FUNCTION LoadDice()
    REM *** Load, scale and place dice ***
    LOAD OBJECT "dice.dbo",diceobj
    POSITION OBJECT diceobj,30,15,45
    SCALE OBJECT diceobj,50,50,50
ENDFUNCTION

FUNCTION PositionCameras()
    AUTOCAM OFF
    REM *** Set up player's camera ***

```

```

MAKE CAMERA 1
SET CAMERA VIEW 1, 0,0,1279,1023
POSITION CAMERA 1,3,5,-52
POINT CAMERA 1,3,2,-40
REM *** Set up dice's camera ***
MAKE CAMERA 2
POSITION CAMERA 2,50,43,0
POINT CAMERA 2,30,15,45
SET CAMERA FOV 2, 10
SET CAMERA VIEW 2,1100,20,1220,120
ENDFUNCTION

FUNCTION MovePlayerToElevator()
  REM *** Move character to lift ***
  LOOP OBJECT playerobj
  x# = OBJECT POSITION X(playerobj)
  y# = OBJECT POSITION Y(playerobj)
  FOR z# = OBJECT POSITION Z(playerobj)
    ↳TO lifts(board(playerat)).zord STEP
    ↳0.2*FindDirectionOfTravel()
    POSITION OBJECT playerobj, x#,y#,z#
    WAIT 50
  NEXT z#
  STOP OBJECT playerobj
  SET OBJECT FRAME playerobj,1
ENDFUNCTION

FUNCTION TurnPlayer(angle)
  IF angle < 0
    stepsize = -1
  ELSE
    stepsize = 1
  ENDIF
  startangle = OBJECT ANGLE Y(playerobj)
  FOR degree = startangle TO startangle
    ↳+ angle STEP stepsize
    YROTATE OBJECT playerobj, degree
    WAIT 1
  NEXT angle
ENDFUNCTION

FUNCTION MoveOntoPlatform()
  LOOP OBJECT playerobj
  x# = OBJECT POSITION X(playerobj)
  y# = OBJECT POSITION Y(playerobj)
  z# = OBJECT POSITION Z(playerobj)
  FOR newx# = x# TO x#-5 STEP -0.1
    POSITION OBJECT playerobj, newx#,
    ↳y#,z#
    WAIT 50
  NEXT moves
  STOP OBJECT playerobj
  SET OBJECT FRAME playerobj,1
ENDFUNCTION

FUNCTION MoveElevator()
  REM *** Get lift's position in lifts ***
  liftnum = board(playerat)
  REM *** Get lift's ID ***
  liftID = lifts(liftnum).ID
  REM *** Calc number of world units to be
  ↳moved ***
  height = lifts(liftnum).floors*10
  REM ***Determine direction (up/down)***
  IF height > 0
    liftdirection = 1
  ELSE
    liftdirection = -1
  ENDIF
  REM *** Get char position ***
  playerx# = OBJECT POSITION X(playerobj)
  playery# = OBJECT POSITION Y(playerobj)
  playerz# = OBJECT POSITION Z(playerobj)
  REM *** Calc player's move per offset
  ↳unit ***
  incmove# = height/100.0
  REM *** IF lift up, final offset 100 ***
  IF liftdirection = 1
    liftendoffset = 100
  ELSE
    liftendoffset = 0
  ENDIF
  REM *** Calculate lift start offset ***
  liftstartoffset = 100 - liftendoffset
  REM *** Move platform, char, and camera ***
  FOR move = liftstartoffset TO
    ↳liftendoffset STEP liftdirection
    OFFSET LIMB liftID,2,0,move,-2
    POSITION OBJECT playerobj,playerx#,
    ↳OBJECT POSITION Y(playerobj)+incmove#
    ↳,playerz#
    POSITION CAMERA 1, playerx#,
    ↳OBJECT POSITION Y(playerobj)+5,
    ↳playerz#-(7*FindDirectionOfTravel())
    WAIT 10
  NEXT move
  playerat = lifts(liftnum).endsquare
ENDFUNCTION liftnum

FUNCTION MoveOffPlatform()
  LOOP OBJECT playerobj
  x# = OBJECT POSITION X(playerobj)
  y# = OBJECT POSITION Y(playerobj)
  z# = OBJECT POSITION Z(playerobj)
  FOR newx# = x# TO x#+5 STEP 0.1
    POSITION OBJECT playerobj, newx#,y#,z#
    WAIT 1
  NEXT moves
  STOP OBJECT playerobj
  SET OBJECT FRAME playerobj,1
ENDFUNCTION

FUNCTION ReturnElevator(liftnum)
  liftID = lifts(liftnum).ID
  height = lifts(liftnum).floors*10
  IF height > 0
    liftdirection = -1
  ELSE
    liftdirection = 1
  ENDIF
  IF liftdirection = -1
    liftendoffset = 0
  ELSE
    liftendoffset = 100
  ENDIF
  liftstartoffset = 100 - liftendoffset
  unitsperperc# = height / 100.0
  FOR move = liftstartoffset TO
    ↳liftendoffset STEP liftdirection
    OFFSET LIMB liftID,2,0,move,-2
    WAIT 10
  NEXT move
ENDFUNCTION

FUNCTION RepositionCamera()
  REM *** Get position of character ***
  x# = OBJECT POSITION X(playerobj)
  y# = OBJECT POSITION Y(playerobj)
  z# = OBJECT POSITION Z(playerobj)
  REM *** IF odd corridor,
  ↳camera -7 z-units ***
  IF FindDirectionOfTravel() = 1
    POSITION CAMERA 1, x#+1,y#+5,z#-7
    POINT CAMERA 1,x#+1,y#+2,z#+5
  ELSE
    REM *** Even corridor, +7 z-units ***
    POSITION CAMERA 1, x#+1,y#+5,z#+7
    POINT CAMERA 1, x#+1,y#+2,z#-5
  ENDIF
ENDFUNCTION

REM *****
REM *** Level 4 ***
REM *****

```

```

FUNCTION AddElevators()
    REM *** Create the first lift ***
    LOAD OBJECT "lift.dbo",firstliftobj
    REM *** Create others as clones ***
    FOR c = 2 TO 27
        CLONE OBJECT lifts(c).ID,firstliftobj
    NEXT c
    REM *** Position all the lifts ***
    FOR c = 1 TO 27
        REM *** Rotate lift and scale to
        ↳match floors ***
        liftID = lifts(c).ID
        YROTATE OBJECT liftID,-90
        SCALE OBJECT liftID,100,
        ↳10*ABS(lifts(c).floors),100
        REM *** place up lifts at start
        ↳square ***
        IF lifts(c).floors> 0
            POSITION OBJECT liftID,-4.5,
            ↳(lifts(c).startsquare-1)/10*10,
            ↳lifts(c).zord
        ELSE
            REM *** And down ones at finish
            ↳square ***
            POSITION OBJECT liftID,-4.5,
            ↳(lifts(c).endsquare-1)/10*10,
            ↳lifts(c).zord
            REM *** down lifts need platform
            ↳at top ***
            OFFSET LIMB liftID,2,0,100,-2
        ENDIF
    NEXT c
ENDFUNCTION

FUNCTION FindDirectionOfTravel()
    REM *** IF odd floor,1 else -1 ***
    IF ((playerat-1)/10) mod 2 = 0
        result = 1
    ELSE
        result = -1
    ENDIF
ENDFUNCTION result

```

Handling BSP Models

Binary Space Partitioning Concepts

PK3 Files

Loading a BSP Model

Camera/BSP Collisions

Object/BSP Collisions

Binary Space Partitioning

Introduction

A Binary Space Partitioning Tree (BSP tree) is a method of storing data about 3D objects. It contains details about the order of the polygons that make up a complex 3D scene, storing information on how the position of polygons relate to each other. This information is stored in a family-tree-type structure (hence the *tree* part of the name) with polygons which lie in front of a specified plane on one side of the tree and those behind the plane on the other side.

Storing the polygons that go to make up a complex 3D model in this way makes the computer's task of culling and rendering much easier, since it can use the information about polygon ordering that is held in the BSP structure to efficiently determine which polygons have to be drawn onto the screen and which are hidden behind other polygons and hence may be ignored.

We could create a BSP file containing a simple object such as a weapon or spacecraft, but it is more typically used to contain whole worlds or levels, with multiple buildings, corridors, rooms, etc.

Although BSP files contain the basic geometry of a world, it does not contain other necessary details such as the images used to texture surfaces. These will be held in separate files.

Often BSP files are stored in an archive file (that is, a file which contains a collection of other files. ZIP and RAR files are also archive files). By using an archive file, all the files required by the model can be stored as a single entity and extracted when required. Such archive files have the extension .PK3.

DarkBASIC Pro has the ability to load files of this format because several major games make use of them, notably Quake and Half-Life, to build the world in which the player is immersed.

You may hear certain terms, such as **world**, **map**, or **level**, used to describe these complex models.

- **World** refers to the complete environment in which the player's character can wander.
- **Map** is often used interchangeably with world, but in fact, a world may consist of several different maps, each map containing the 3D elements of a particular section of a world.
- **Level** is a term originally derived from the old game of Dungeons and Dragons which would often take place in a dungeon with sets of stairs leading down to lower and lower levels. Although it is often used in the same context, with a level consisting of rooms, corridors and stairs, it is also used more generally to refer to different areas or different levels of difficulty within a game.

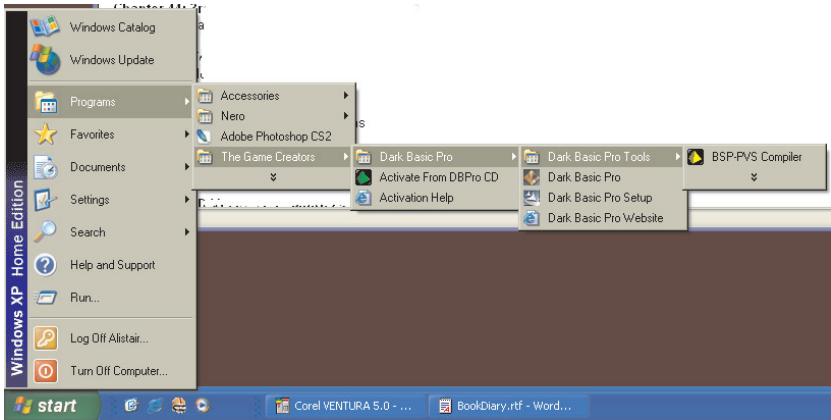
It is possible to use the level editor which comes with a commercial game to create your own playing environment and then import the resulting BSP file into your DarkBASIC Pro program.

Creating a BSP File

Although it's most likely that a level-editor such as the one that comes free with Half-Life will be used to create a BSP world, any normal DirectX file (.X file) can be converted to BSP format using the BSP compiler that comes with DarkBASIC Pro (see FIG-43.1).

FIG-43.1

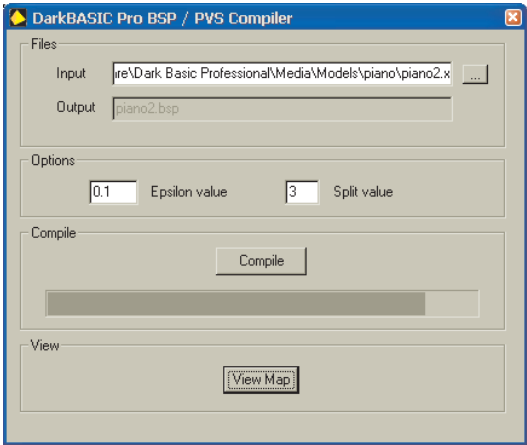
Accessing the BSP Compiler



In the dialog box that the application creates, enter the name of the .X file to be converted (see FIG-43.2).

FIG-43.2

Specifying the File to be Converted



This program creates only the BSP file, not the archive file containing related texture images used by the model.

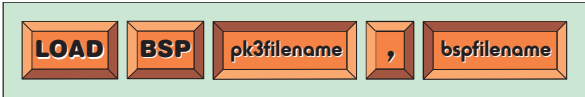
Using BSP Files

The LOAD BSP Statement

BSP files are usually held, along with any texture files required, in a compressed file with a .PK3 extension. To extract the BSP model from the file we use the LOAD BSP statement which has the format shown in FIG-43.3.

FIG-43.3

The LOAD BSP Statement



In the diagram:

pk3filename

is a string giving the name of the archive file containing the BSP file. This string may contain path information.

bspfilename

is a string giving the name of the BSP file held within the PK3 file. Only one BSP model can be in use at any time.

For example, if the file *world.pk3* contains the file *castle.bsp* then, assuming the file is in the current project's folder, we could load the castle model using the line:

```
LOAD BSP "world.pk3", "castle.bsp"
```

The castle model will be loaded and the textures, which should be in the *world.pk3* file, will be applied automatically to the model.

When a BSP file has not been archived within a PK3 file, then *pk3filename* should be an empty string. For example, if the BSP file *dungeon.bsp* is in the current project's folder, we could load the model using the line:

```
LOAD BSP "", "dungeon.bsp"
```

With no PK3 file to search for the texture images, DarkBASIC Pro will look for those images in the current directory. So, if the model in *dungeon.bsp* uses texture images *cobblesstones.jpg* and *stonewall.jpg*, these two files must be in the current project folder along with *dungeon.bsp*.

A loaded BSP model cannot be repositioned, resized, or rotated. Only a single BSP file can be loaded.

The program in LISTING-43.1 loads the BSP file that comes in one of the examples packaged with DarkBASIC Pro.

LISTING-43.1

Loading a BSP Model

```
REM *** Main section ***
ScreenSetUp()

REM *** Load BSP ***
LOAD BSP "", "ikzdm1.bsp"

REM *** Use camera to move around ***
DO
    CONTROL CAMERA USING ARROWKEYS 0,0.1,0.1
LOOP

REM *** End Program ***
WAIT KEY
END

FUNCTION ScreenSetUp()
    SET DISPLAY MODE 1280,1024,32
    COLOR BACKDROP RGB(255,255,100)
    BACKDROP ON
    AUTOCAM OFF
    POSITION CAMERA 0,10,-30
    POINT CAMERA 0,0,0
ENDFUNCTION
```

The *ikzdm1.bsp* file can be found in DarkBASIC Pro's *Dark Basic Software\Dark Basic Professional\Help\Tutorials\World* folder.

Activity 43.1

Type in the program given in LISTING-43.1 (*bsp01.dbpro*).

Before you try running the program, copy the BSP file *ikzdm1.bsp*, and the textures it requires, *CYGRASS.bmp* and *CYWALLT.bmp*, into the project's folder.

What happens when you attempt to move through the wall?

The SET BSP CAMERA COLLISION Statement

To stop the camera moving through parts of our model, we need to use the SET BSP CAMERA COLLISION statement. This statement has the format shown in FIG-43.4.

FIG-43.4

The SET BSP CAMERA COLLISION Statement



In the diagram:

collindex

is an integer value giving an ID to the collision details.

camno

is an integer value identifying the camera whose collision properties are to be set.

radius

is a real number giving the radius of the collision sphere around the camera.

response

-1, 0, or 1. This value determines how the camera reacts to a collision.

-1 : collision ignored (the camera still passes through solid objects).

0 : the camera slides along the object hit.

1 : the camera halts when a collision occurs.

Activity 43.2

Add a SET BSP CAMERA COLLISION statement to your previous program immediately before the DO statement.

Set the *collindex* value to 1; *camno* to zero; *radius* to 2; and *response* to 1.

What happens this time when you attempt to move the camera through the wall (approach the wall at an angle, rather than straight on)?

Find out how the camera reacts when *response* is set to -1 and 0.

The SET BSP OBJECT COLLISION Statement

Just as it is necessary to specify BSP/camera collision characteristics, so we also

need to set the characteristic for BSP/object collisions. This is done using the SET BSP OBJECT COLLISION statement which has the format shown in FIG-43.5.

FIG-43.5

The SET BSP OBJECT COLLISION Statement



In the diagram:

collindex

is an integer value giving an ID to the collision details.

objno

is an integer value identifying the object whose collision properties are to be set.

radius

is a real number giving the radius of the collision sphere around the object.

response

-1, 0, or 1. This value determines how the object reacts to a collision.

-1 : collision ignored (the object still passes through solid objects).

0 : the object slides along the object hit.

1 : the object halts when a collision occurs.

The program in LISTING-43.2 shows a sphere hitting the courtyard floor for each possible value of *response*.

LISTING-43.2

BSP/Object Collision

```

ScreenSetUp()
REM *** Load BSP ***
LOAD BSP "", "ikzdm1.bsp"
REM *** Create sphere ***
MAKE OBJECT SPHERE 1, 10
FOR response = -1 TO 1
  POSITION OBJECT 1, -10, 20, -5
  POINT OBJECT 1, 0, -5, 0
  SET BSP OBJECT COLLISION 1, 1, 2, response
  time = TIMER()
  WHILE TIMER() - time < 7000
    MOVE OBJECT 1, 1
  ENDWHILE
NEXT c
REM *** End Program ***
END

FUNCTION ScreenSetUp()
  SET DISPLAY MODE 1280, 1024, 32
  COLOR BACKDROP RGB(255, 255, 100)
  BACKDROP ON
  AUTOCAM OFF
  POSITION CAMERA 0, 35, -80
  POINT CAMERA 5, 0, 0
ENDFUNCTION

```

Activity 43.3

Type in and test the program in LISTING-43.2 (*bsp02.dbpro*).

Activity 43.4

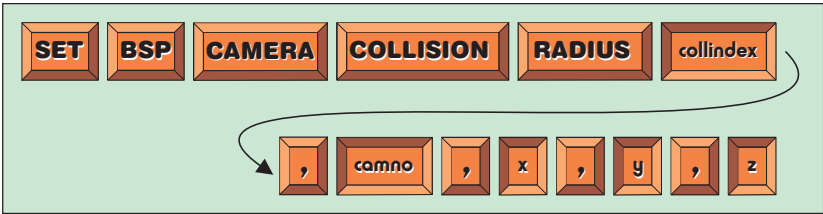
Write a program (*Act4304.dbpro*) which loads *ikzdm1.bsp*. It should then load the Hivebrain walking model (scaling it 200% in each dimension), placing it at (0,-32,0).

The model should have a collision radius of 2 and implement sliding collision. Point the alien at (-50,-32,100) and start it walking. Keep the camera on the model at all times.

The SET BSP CAMERA COLLISION RADIUS Statement

Although we must specify the collision sphere's radius when using the SET BSP CAMERA COLLISION statement, that sphere can be modified and distorted using the SET BSP CAMERA COLLISION RADIUS statement. The statement has the format given in FIG-43.6.

FIG-43.6
The SET BSP CAMERA COLLISION RADIUS Statement



In the diagram:

- collindex* is an integer value giving an ID to the collision details.
- camno* is an integer value identifying the camera whose collision volume properties are to be set.
- x,y,z* is a sequence of real values giving the radii of the ovoid along the x, y and z axes.

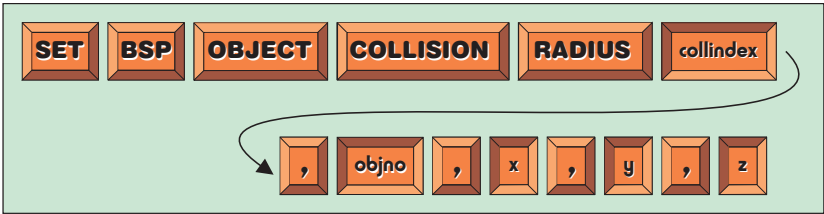
Since the radius of the collision volume can be adjusted separately in all three dimensions, it need no longer be spherical in shape. For example, we could change the collision volume for camera zero to an ovoid shape using the line:

```
SET BSP CAMERA COLLISION RADIUS 1,0,5,10,5
```

The SET BSP OBJECT COLLISION RADIUS Statement

Similarly, the collision volume of an object can be set using the SET BSP OBJECT COLLISION RADIUS statement which has the format shown in FIG-43.7.

FIG-43.7
The SET BSP OBJECT COLLISION RADIUS Statement



In the diagram:

collindex

is an integer value giving an ID to the collision details.

objno

is an integer value identifying the object whose collision volume properties are to be set.

x,y,z

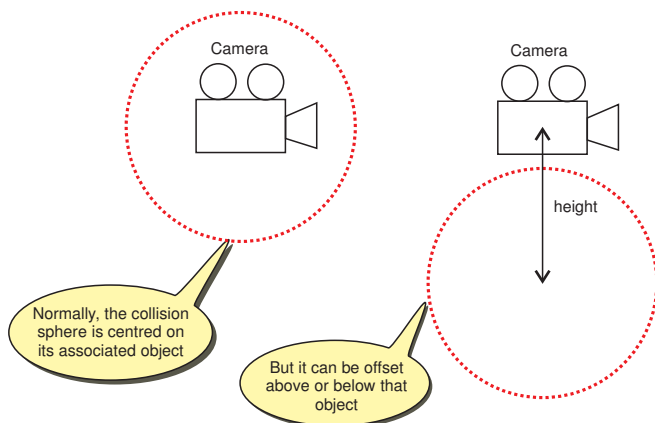
is a sequence of real values giving the radii of the ovoid along the x, y and z axes.

The SET BSP COLLISION HEIGHT ADJUSTMENT Statement

Normally a collision sphere is centred on the camera or object to which it is attached, but it is possible to reposition this sphere either above or below this normal position (see FIG-43.8).

FIG-43.8

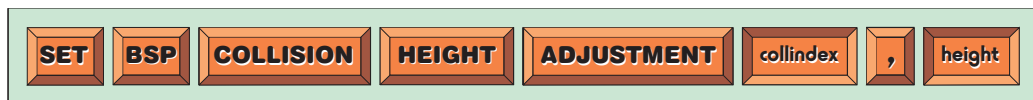
A Repositioned Collision Sphere



Offsetting the collision sphere is achieved using the SET BSP COLLISION HEIGHT ADJUSTMENT statement which has the format given in FIG-43.9.

FIG-43.9

The SET BSP COLLISION HEIGHT ADJUSTMENT Statement



In the diagram:

collindex

is an integer value of the collision ID previously assigned.

height

is a real number giving the height of the collision sphere from the camera. A negative value means that the collision sphere is below the camera.

For example, if we had previously set the camera's collision details using the line

```
SET BSP CAMERA COLLISION 1,0,2,1
```

we could move the camera's collision sphere far below ground level using the line

```
SET BSP COLLISION HEIGHT ADJUSTMENT 1,-50
```

and since the collision sphere is no longer anywhere near the BSP model, the camera will be able to move through the walls once more.

Activity 43.5

Add the above statement to your program *bsp01.dbpro* (after the SET BSP CAMERA POSITION statement), and check that the camera can again move through the walls. Remove the statement after testing the program.

The SET BSP COLLISION THRESHOLD Statement

When a sliding collision causes an object or a camera to move downwards, the sensitivity of the movement can be adjusted using the SET BSP COLLISION THRESHOLD statement which has the format shown in FIG-43.10.

FIG-43.10

The SET BSP COLLISION THRESHOLD Statement



In the diagram:

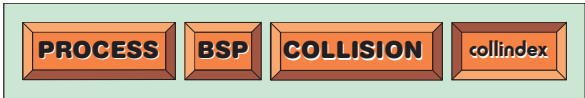
- collindex* is an integer value of the collision ID previously assigned.
- sensitivity* is a real number giving the sensitivity required to initiate the sliding fall. The default value is zero, which causes falls to happen due to the slightest change in surface height. A higher value will delay the fall until the object/camera is further from the new ground level.

The PROCESS BSP COLLISION Statement

Collisions are tested for automatically each time the screen is refreshed, but it is possible for detection at an earlier point by calling the PROCESS BSP COLLISION statement which has the format shown in FIG-43.11.

FIG-43.11

The PROCESS BSP COLLISION Statement



In the diagram:

- collindex* is an integer value of the collision ID previously assigned.

The SET BSP COLLISION OFF Statement

Detection of a specific BSP collision (camera or objects) can be switched off using the SET BSP COLLISION OFF statement which has the format shown in FIG-43.12.

FIG-43.12

The SET BSP COLLISION OFF Statement



In the diagram:

collindex is an integer value specifying the ID of the collision to be switched off.

Activity 43.6

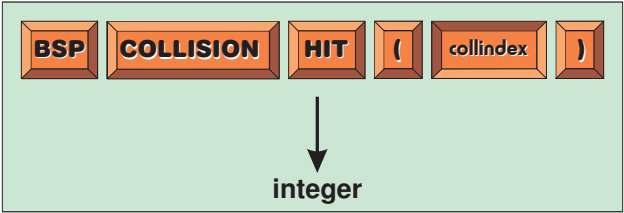
Modify *bsp01.dbpro* so that the BSP/camera collision detection is switched off.

The BSP COLLISION HIT Statement

We can detect if a camera/object has collided with the BSP model using the BSP COLLISION HIT statement which has the format shown in FIG-43.12.

FIG-43.12

The BSP COLLISION HIT Statement



In the diagram:

collindex is an integer value of the collision ID previously assigned.

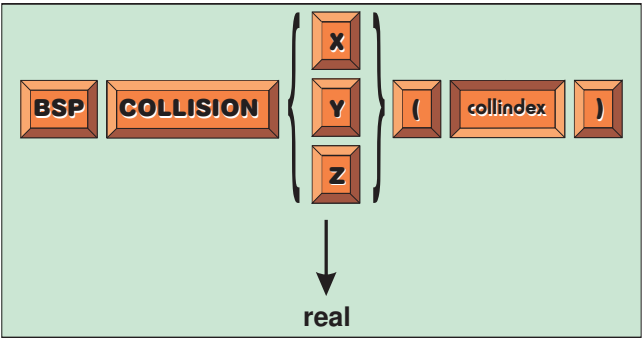
The statement returns 1 if a collision has occurred, otherwise zero is returned.

The BSP COLLISION Statement

The coordinates of a collision can be discovered using the BSP COLLISION statement which has the format shown in FIG-43.13.

FIG-43.13

The BSP COLLISION Statement



In the diagram:

X,Y,Z Use the option appropriate for the ordinate required.

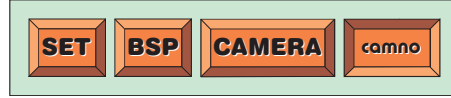
collindex is an integer value of the collision ID previously assigned.

The SET BSP CAMERA Statement

Normal cameras (as used in programs not containing BSP models) do not operate correctly with BSP objects. Usually this won't be a problem, since camera zero is automatically modified when a BSP model is loaded, but if we want to use a second camera, then it needs to be converted to a BSP camera. This can be done using the SET BSP CAMERA statement which has the format shown in FIG-43.14.

FIG-43.14

The SET BSP
CAMERA Statement



In the diagram:

camno

is an integer value giving the ID of the camera to be converted to a BSP camera.

The DELETE BSP Statement

A BSP model can be removed from RAM using the DELETE BSP statement which has the format shown in FIG-43.15.

FIG-43.15

The DELETE BSP
Statement



Once this statement has been executed a new BSP model may be loaded.

The SET BSP MULTITEXTURING Statement

Some BSP models may have been created using multiple overlapping textures (known as multitexturing). When recreated on screen, the visual effect created by multitexturing can be switched on or off using the SET BSP MULTITEXTURING statement which has the format shown in FIG-43.16.

FIG-43.16

The SET BSP
MULTITEXTURING
Statement



Summary

- Binary Space Partitioning (BSP) is a method of storing 3D models designed to minimise the number of polygons that have to be processed when recreating a 3D model on the screen.
- BSP files and the texture they use are often archived in a PK3 file.
- BSP files have a .BSP extension.
- DarkBASIC Pro comes with a utility for converting .X files to BSP files.
- BSP files are commonly used for creating level, map or world models.
- Use LOAD BSP to load a BSP file.

- BSP models cannot be moved, resized or rotated.
- Only one BSP model can be loaded.
- Use SET BSP CAMERA COLLISION to stop the camera moving through a BSP model.
- Use SET BSP OBJECT COLLISION to stop another 3D object passing through a BSP model.
- Use SET BSP CAMERA COLLISION RADIUS to set the radius of a camera's collision sphere.
- Use SET BSP OBJECT COLLISION RADIUS to set the radius of an object's collision sphere.
- Use SET BSP COLLISION HEIGHT ADJUSTMENT to reposition a camera or object's collision sphere.
- Use SET BSP COLLISION THRESHOLD to set the sensitivity of a sliding collision.
- Use PROCESS BSP COLLISION to initiate early collision detection.
- Use SET BSP COLLISION OFF to stop detection of BSP collisions.
- Use BSP COLLISION HIT to detect if a camera or object has collided with the BSP model.
- Use BSP COLLISION to determine the coordinates at which the collision occurred.
- Use SET BSP CAMERA to convert additional cameras for use with a BSP model.
- Use DELETE BSP to delete a BSP model from RAM.
- Use SET BSP MULTITEXTURING to allow the surfaces of a BSP model to use multiple texture files.

Using a BSP Map

Introduction

This section demonstrates the use of a BSP map as the environment for a first-person shooter. Your invisible character is equipped with a pistol and an endless supply of bullets.

Notice how sound effects have been added to increase realism. A simple trick has been added to give the impression of the bullet being vapourised as it hits the wall.

The Program

The main logic of the program is:

```
Set up screen
Initialise game
DO
    Control player
    IF BulletFired THEN
        Move bullet
    ENDIF
LOOP
```

This is coded as:

```
SetUpScreen()
InitialiseGame()
DO
    ControlPlayer()
    IF BulletFired()
        MoveBullet()
    ENDIF
LOOP
END
```

The main routines are:

SetUpScreen	This routine initialises the screen resolution and positions the camera. It also hides the mouse pointer.
InitialiseGame	This routine loads all the BSP maps, models, and sounds. It also locks the gun to the screen giving the typical FPS effect.
ControlPlayer	This routine allows the player to move the camera using the arrow keys and also calls <i>FireGun()</i> if the Enter key is pressed.
FireGun	This routine fires the bullet object, positioning the bullet beside the gun and playing the firing sound.
BulletFired	This routine returns 1 if a bullet is currently in flight; otherwise zero is returned.

MoveBullet

This routine moves the bullet 2 units and checks for a collision with the BSP world. If a hit is detected, *ShowBulletHit()* is called.

ShowBulletHit

This routine creates the vapourised bullet effect, hides the bullet object and plays a *bullet hit* sound.

The code for the program is given in LISTING-43.3.

LISTING-43.3

Using a BSP Map

```
REM *** Object Constants ***
#CONSTANT gunobj 1
#CONSTANT bulletobj 2
REM *** Sound constants ***
#CONSTANT firesnd 1
#CONSTANT hitsnd 2

SetUpScreen()
InitialiseGame()
DO
    ControlPlayer()
    IF BulletFired()
        MoveBullet()
    ENDIF
LOOP
END

FUNCTION SetUpScreen()
    REM *** Set screen resolution ***
    SET DISPLAY MODE 1280,1024,32
    BACKDROP OFF
    REM *** Hide mouse ***
    HIDE MOUSE
    REM *** Position camera ***
    AUTOCAM OFF
    POSITION CAMERA 2,2,2
    POINT CAMERA 0,0,0
ENDFUNCTION

FUNCTION InitialiseGame()
    REM *** Load BSP map ***
    LOAD BSP "", "ikzdml.bsp"
    REM *** Set up collision parameters ***
    SET BSP CAMERA COLLISION 1,0,1,0
    REM *** Load the weapon ***
    LOAD OBJECT "H-Pistol-Static.x", gunobj
    REM *** Position object in front of camera ***
    LOCK OBJECT ON gunobj
    SCALE OBJECT gunobj, 300, 300, 300
    POSITION OBJECT gunobj, 0, -1, 2
    REM *** Create hidden object to use as bullet ***
    MAKE OBJECT SPHERE bulletobj, 0.1
    SET BSP OBJECT COLLISION bulletobj, 2, 0.1, 0
    HIDE OBJECT bulletobj
    REM *** Load sound file ***
    LOAD SOUND "gun 2.wav", gunobj
    LOAD SOUND "mortar.wav", hitsnd
ENDFUNCTION

FUNCTION ControlPlayer()
    REM *** Control camera ***
    CONTROL CAMERA USING ARROWKEYS 0,0.5,0.5
    REM *** IF Enter pressed, attempt to fire ***
    IF INKEY$( ) = CHR$(13)
        FireGun()
    ENDIF
ENDFUNCTION
```

LISTING-43.3

(continued)

Using a BSP Map

```
FUNCTION FireGun()
    REM *** If bullet, visible, exit ***
    IF BulletFired()
        EXITFUNCTION
    ENDIF
    REM *** Position and show bullet ***
    SHOW OBJECT bulletobj
    POSITION OBJECT bulletobj,CAMERA POSITION X(),
    ↵CAMERA POSITION Y()-2,CAMERA POSITION Z()
    ROTATE OBJECT bulletobj,CAMERA ANGLE X(),
    ↵CAMERA ANGLE Y(),CAMERA ANGLE Z()
    REM *** Play sound of gun fire ***
    PLAY SOUND firesnd
ENDFUNCTION

FUNCTION BulletFired()
    REM *** Bullet travelling if visible ***
    result = OBJECT VISIBLE(bulletobj)
ENDFUNCTION result

FUNCTION MoveBullet()
    REM *** Move bullet ***
    MOVE OBJECT bulletobj,2
    REM *** IF it hits wall,show wall being it ***
    IF BSP COLLISION HIT(bulletobj)
        ShowBulletHit()
    ENDIF
ENDFUNCTION

FUNCTION ShowBulletHit()
    REM *** Make bullet larger and brighter ***
    SCALE OBJECT bulletobj, 1000,1000,1000
    SET OBJECT EMISSIVE bulletobj, RGB(255,255,255)
    REM *** Show in this state for 10 millisecs ***
    WAIT 10
    REM *** Hide bullet and darken ***
    HIDE OBJECT bulletobj
    SET OBJECT EMISSIVE bulletobj, RGB(100,100,100)
    REM *** Return to normal size ***
    SCALE OBJECT 2, 100,100,100
    REM *** Play hit sound ***
    PLAY SOUND hitsnd
ENDFUNCTION
```

Activity 43.7

Type in and test the program in LISTING-43.3 (*bsp03.dbpro*).

Solutions

Activity 43.1

The camera can move straight through the walls.

Activity 43.2

```
ScreenSetUp()  
REM *** Load BSP ***  
LOAD BSP "", "ikzdm1.bsp"  
SET BSP CAMERA COLLISION 1,0,2,1  
REM *** Use camera to move around ***  
DO  
    CONTROL CAMERA USING ARROWKEYS 0,0.1,0.1  
LOOP  
REM *** End Program ***  
WAIT KEY  
END  
  
FUNCTION ScreenSetUp()  
    SET DISPLAY MODE 1280,1024,32  
    COLOR BACKDROP RGB(255,255,100)  
    BACKDROP ON  
    AUTOCAM OFF  
    POSITION CAMERA 0,10,-30  
    POINT CAMERA 0,0,0  
ENDFUNCTION
```

This time the camera is stopped when it meets the wall.

Using a value of -1 allows the camera to travel through the wall.

A value of zero creates a sliding collision effect when the camera comes into contact with the wall.

Activity 43.3

No solution required.

Activity 43.4

```
ScreenSetUp()  
REM *** Load BSP ***  
LOAD BSP "", "ikzdm1.bsp"  
REM *** Load Hivebrain ***  
LOAD OBJECT "H-Alien Hivebrain-Move.x",1  
SCALE OBJECT 1, 200,200,200  
POSITION OBJECT 1,0,-32,0  
SET BSP OBJECT COLLISION 1,1,2,0  
POINT OBJECT 1,-50,-32,100  
LOOP OBJECT 1  
DO  
    MOVE OBJECT 1,0.1  
    POINT CAMERA OBJECT POSITION X(1),OBJECT  
POSITION Y(1),OBJECT POSITION Z(1)  
LOOP  
REM *** End Program ***  
END  
  
FUNCTION ScreenSetUp()  
    SET DISPLAY MODE 1280,1024,32  
    COLOR BACKDROP RGB(255,255,100)  
    BACKDROP ON  
    AUTOCAM OFF  
    POSITION CAMERA -120,-20,160  
    POINT CAMERA 100,0,0  
ENDFUNCTION
```

Activity 43.5

The main section changes to:

```
ScreenSetUp()  
REM *** Load BSP ***  
LOAD BSP "", "ikzdm1.bsp"  
SET BSP CAMERA COLLISION 1,0,2,1  
SET BSP COLLISION HEIGHT ADJUSTMENT 1,-50  
REM *** Use camera to move around ***  
DO  
    CONTROL CAMERA USING ARROWKEYS 0,0.1,0.1  
LOOP  
REM *** End Program ***  
WAIT KEY  
END
```

Activity 43.6

The main section should be coded as:

```
ScreenSetUp()  
REM *** Load BSP ***  
LOAD BSP "", "ikzdm1.bsp"  
SET BSP CAMERA COLLISION 1,0,2,1  
SET BSP COLLISION OFF 1  
REM *** Use camera to move around ***  
DO  
    CONTROL CAMERA USING ARROWKEYS 0,0.1,0.1  
LOOP  
REM *** End Program ***  
WAIT KEY  
END
```

Activity 43.7

No solution required.

Creating Terrain

Advanced Terrain Statements

Developing a Terrain-Based Game

Finding the Height of a Point on a Terrain Object

Texturing a Terrain Object

Using a Greyscale Image to Contour a Landscape

Using TERRAIN Statements to Create a Landscape

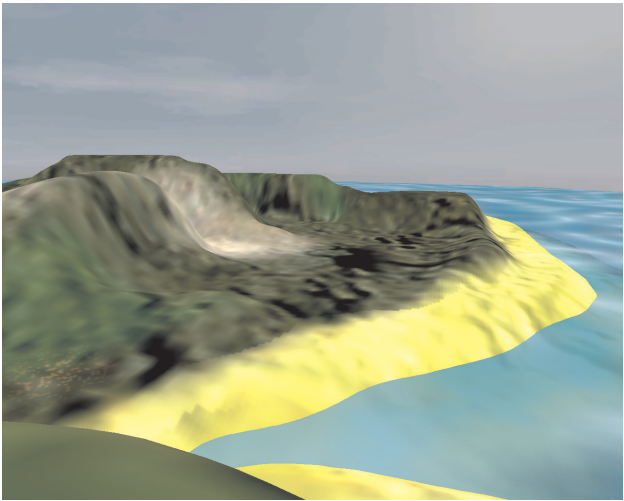
Creating Terrain

Introduction

One way to create undulating, ground-like surface is to use the TERRAIN statements. This approach allows you to use a greyscale image to determine the contours of the terrain with black areas in the image being at height zero and white areas in the image corresponding to the maximum height. Shades of grey in the image create intermediate heights. An example of the effects that can be achieved is shown in FIG-44.1.

FIG-44.1

Creating Terrain



The documented terrain commands are few in number, but several newer statements which are currently undocumented are also available. We'll start with the documented statements.

Documented Terrain Statements

The MAKE TERRAIN Statement

A terrain object is created using the MAKE TERRAIN statement. This assigns an integer value to the terrain and links it to a greyscale image. It is this image that determines the contours of the terrain. The MAKE TERRAIN statement has the format shown in FIG-44.2

FIG-44.2

The MAKE TERRAIN Statement



In the diagram:

terno

is the integer value assigned to the terrain object being created.

filename

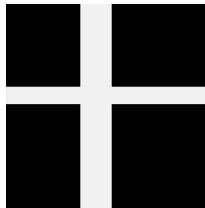
is a string giving the name of the greyscale file used to determine the contours of the terrain object. The file must be square, with its width and height being equal.

The terrain's dimensions will change to match those of the image being used as the contour data. For example, a 512 by 512 image will create a terrain object which is exactly 512 by 512. Initially, the bottom-left corner of the terrain is at position (0,0,0).

The program in LISTING-44.1 uses image shown in FIG-44.3 to control the contours of the terrain created.

FIG-44.3

Image Used to
Contour Terrain



Since the image used in this example contains only black and white (no shades of grey) the terrain contours will be flat (black) or at maximum height (white).

The program changes the default light to a point light in order to produce shadows over the terrain. The position of the camera can be modified using the arrow keys.

LISTING-44.1

Creating a Terrain Object

```

SetUpScreen()

REM *** Create terrain ***
MAKE TERRAIN 1,"contour01.bmp"

REM *** Allow user to move camera ***
DO
    CONTROL CAMERA USING ARROWKEYS 0,5,1
LOOP

REM *** End program ***
END

FUNCTION SetUpScreen()
    REM *** Set display mode required ***
    SET DISPLAY MODE 1280,1024,32
    REM *** position camera ***
    AUTOCAM OFF
    POSITION CAMERA 0,100,200,-100
    REM *** Change to point light ***
    SET POINT LIGHT 0,0,100,500
    SET AMBIENT LIGHT 50
ENDFUNCTION
    
```

Activity 44.1

Type in and test the program given above (*terrain01.dbpro*).

Change the contour image used to *contour02.bmp*.

Create your own greyscale image (it must be square) and use it in your program to define the contours of the terrain. (Return to using *contour02.bmp* once you've tested your own image.)

The contouring works best if the image's dimensions are a power of 2. For example, 256x256, 512x512, or 1024x1024.

The DELETE TERRAIN Statement

When a terrain object is no longer required by a program, it should be deleted using

the DELETE TERRAIN statement which has the format shown in FIG-44.4.

FIG-44.4

The DELETE TERRAIN
Statement



In the diagram:

terno

is the integer value previously assigned to terrain object.

This will free up the RAM space occupied by the terrain object.

The POSITION TERRAIN Statement

Initially, the terrain is placed in 3D space with its bottom-left corner at position (0,0,0) and the top-right corner at point (x,0,x) where *x* is the width of the contour image. For example, if you are using a 512 x 512 greyscale image to supply the contour information, then the terrain's top-right corner will be positioned at (512,0,512).

If you need to move the terrain elsewhere within the 3D world, you can use the POSITION TERRAIN statement which has the format shown in FIG-44.5.

FIG-44.5

The POSITION TERRAIN
Statement



In the diagram:

terno

is the integer value previously assigned to the terrain object.

x,y,z

is a sequence of 3 real numbers giving the new position for the top-left corner of the terrain.

For example, we could move terrain object 1 so that its top-left corner is at position (200,100,-50) using the statement:

```
POSITION TERRAIN 1,200,100,-50
```

Activity 44.2

Modify your last program to reposition the centre of your terrain to location (0,0,0).
(HINT: You'll need to load the contour image as a bitmap and get its dimensions.)

Reposition the camera so that it is at the centre of the terrain and at a height of 50.

Copy the *camera.dba* file we created in Chapter 33 into the current project folder.

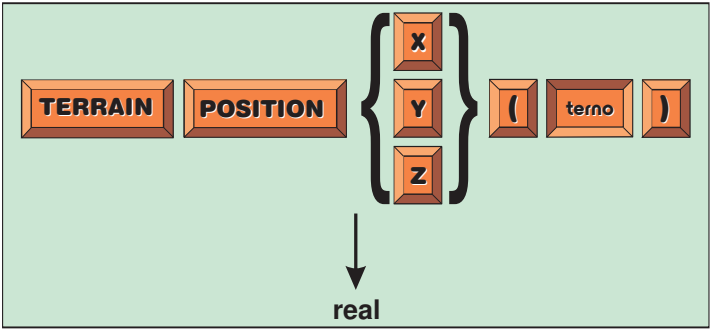
Change camera control to use *PointCameraUsingMouse()* and *MoveCameraUsingMouse()*.

The TERRAIN POSITION Statement

The x, y or z ordinate of the top-left corner of a terrain object is returned using the TERRAIN POSITION statement which has the format shown in FIG-44.6.

FIG-44.6

The TERRAIN POSITION Statement



In the diagram:

X,Y,Z

Use the appropriate option for the ordinate required.

terno

is the integer value previously assigned to the terrain object whose position is required.

The statement returns a real number representing the ordinate requested.

For example, to find the coordinates of the top-left corner of terrain object 1, we could use the statement:

```
xord# = TERRAIN POSITION X(1)
yord# = TERRAIN POSITION Y(1)
zord# = TERRAIN POSITION Z(1)
```

Activity 44.3

What statement would be used to display the y ordinate of terrain object 2?

The TEXTURE TERRAIN Statement

As with other 3D objects, you can add a texture to the terrain by linking a loaded image to the terrain. This is achieved using the TEXTURE TERRAIN statement which takes the format shown in FIG-44.7.

FIG-44.7

The TEXTURE TERRAIN Statement



In the diagram:

terno

is the integer value previously assigned to terrain object.

imgno

is the integer value previously assigned to the image object that is to be used to texture the terrain.

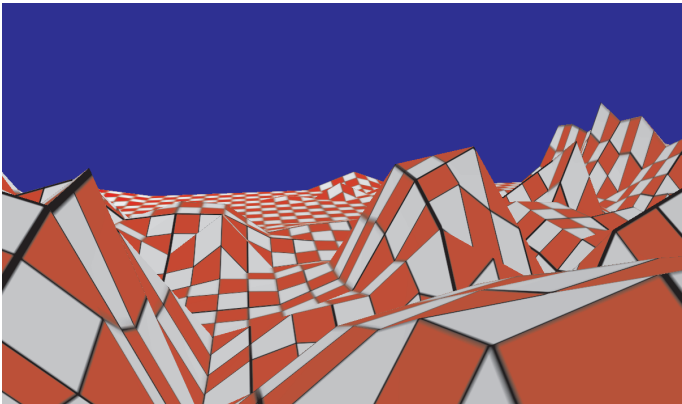
For example, to use the image *grid8by8.bmp* to texture terrain object 1, we could use the following code:

```
LOAD IMAGE "grid8by8.bmp",2
TEXTURE TERRAIN 1,2           `Texture terrain 1 with image 2
```

The result is shown in FIG-44.8.

FIG-44.8

A Textured Terrain



The image used will be tiled automatically if necessary. Make sure a seamless image is used and the tiling will be invisible.

Activity 44.4

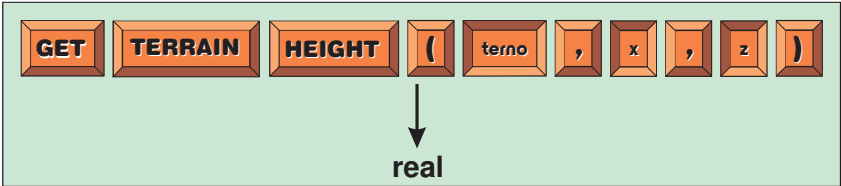
Add an image object (containing the file *grass256.jpg*) to your previous program and use this to texture the terrain.
Can the camera move through the terrain?

The GET TERRAIN HEIGHT Statement

With other 3D objects we could use the AUTOMATIC CAMERA COLLISION statement to prevent the camera moving through what should be solid 3D objects. Unfortunately, that command does not work with terrain objects. So to solve this problem we need to make sure that the camera stays above the surface at all times. We can do this using the GET TERRAIN HEIGHT statement which returns the height of the terrain at a specific point and adjusting the camera's position accordingly. The GET TERRAIN HEIGHT statement has the format shown in FIG-44.9.

FIG-44.9

The GET TERRAIN HEIGHT Statement



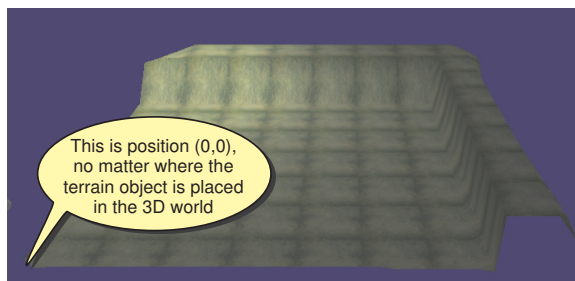
In the diagram:

- terno* is the integer value previously assigned to the terrain object.
- x,z* are the coordinates of the position within the terrain whose height is to be returned.

The statement returns a real number representing the y-ordinate value at position (x,z) on terrain *terno*. The bottom-left corner of the terrain is taken as position (0,0) irrespective of the terrain object's own position within the 3D world (see FIG-44.10).

FIG-44.10

The Terrain Origin



For example, we could return the height of position (10,20) within terrain 1 using the line:

```
height# = GET TERRAIN HEIGHT(1,10,20)
```

The program in LISTING-44.2 moves a small cube across the terrain (from left to right). It attempts to keep the cube half buried in the terrain by using the GET TERRAIN HEIGHT statement. The camera follows the movement.

LISTING-44.2

Using the GET
TERRAIN HEIGHT
Statement

```
SetUpScreen()
REM *** Create a terrain object ***
MAKE TERRAIN 1,"terral.bmp"
REM *** Texture terrain ***
LOAD IMAGE "grass.jpg",1
TEXTURE TERRAIN 1,1
REM *** Match terrain coordinates to real coordinates ***
POSITION TERRAIN 1,0,0,511
REM *** Make and position cube ***
MAKE OBJECT CUBE 1,10
POSITION OBJECT 1, 0,0,50
height# = GET TERRAIN HEIGHT(0,0,50)
REM *** Move cube across terrain ***
FOR c = 0 TO 511
    REM *** Display current height ***
    SET CURSOR 10,10
    PRINT "CURRENT HEIGHT :",height#
    REM *** Recalculate cube's height and reposition camera ***
    height# = GET TERRAIN HEIGHT(1,c,50)
    POSITION OBJECT 1,c,height#,50
    POSITION CAMERA c-5,height#+10,0
    POINT CAMERA c,height#,50
    WAIT 10
NEXT c
REM *** End program ***
WAIT KEY
END

FUNCTION SetUpScreen()
    REM *** Set display mode required ***
    SET DISPLAY MODE 1280,1024,32
    COLOR BACKDROP RGB(255,255,100)
    BACKDROP ON
    REM *** position camera ***
    AUTOCAM OFF
    POSITION CAMERA 0,0,10,-100
    REM *** Change to point light ***
    SET POINT LIGHT 0,0,100,500
    SET AMBIENT LIGHT 50
ENDFUNCTION
```

AUTHOR'S NOTE: In the very few tests I have tried using this statement, I find that multiplying the value returned by 0.65 gives a more accurate estimate of the height.

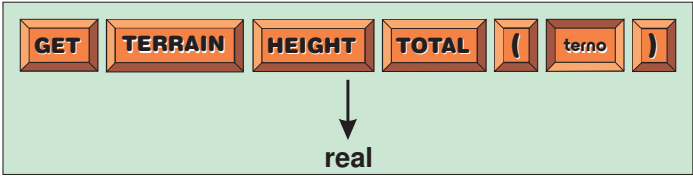
Activity 44.5

Type in and test the program given in LISTING-44.2 (*terrain02.dbpro*).
Is the height value accurate?

The GET TOTAL TERRAIN HEIGHT Statement

The height of the highest point on a terrain object can be discovered using the GET TOTAL TERRAIN HEIGHT statement which has the format shown in FIG-44.11.

FIG-44.11
The GET TOTAL
TERRAIN HEIGHT
Statement



In the diagram:

terno is the integer value previously assigned to the terrain object.

AUTHOR'S NOTE:
This statement seems to
return the value 100 for
most terrain maps.

For example, we could find the highest point on terrain 1 using the statement:

```
highest# = GET TOTAL TERRAIN HEIGHT TOTAL (1)
```

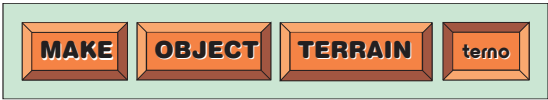
The Advanced Terrain Statements

The advanced terrain statements (which are currently undocumented) offer not only alternative ways of creating a terrain, but also extra statements which allow us more control over the terrains we create. As a general rule, these newer statements should be considered as replacements for the older terrain statements.

The MAKE OBJECT TERRAIN Statement

An alternative way of creating a terrain object is to use the MAKE OBJECT TERRAIN statement which has the format shown in FIG-44.12.

FIG-44.12
The MAKE OBJECT
TERRAIN Statement



In the diagram:

terno is an integer value specifying the ID to be assigned to the terrain object being created.

To create terrain object 1 we would use the line:

```
MAKE OBJECT TERRAIN 1
```

The terrain created in this way is a normal 3D object (just like a cube or sphere) and can be operated on by most 3D statements such as POSITION OBJECT.

By itself, MAKE OBJECT TERRAIN is of little use, since it does not create a visible terrain. We need to add a few more statements...

The SET TERRAIN HEIGHTMAP Statement

Once a terrain object has been created, we need to specify its contour map. This is done using the SET TERRAIN HEIGHTMAP statement which has the format shown in FIG-44.13.

FIG-44.13

The SET TERRAIN
HEIGHT MAP



In the diagram:

<i>terno</i>	is an integer value identifying the terrain whose contour map is to be loaded.
<i>filename</i>	is a string giving the name of the contour map file. The string may include path information. Any file of 256 or more colours can be used.

For example, assuming we want to use the file *terra1.bmp* as our contour file for terrain 1 and that the file is in the project directory, then we could use the statement:

SET TERRAIN HEIGHT MAP 1, "terra1.jpg"

As you know the pixels that make up an image are represented as a sequence of numbers. It is these numbers that determine the height of any point on the terrain object; low numbers give low points, high numbers create high points.

The SET TERRAIN SCALE Statement

The third stage in the process of creating a terrain is scaling. By default, a terrain will be sized to match the dimensions of its contour map. Therefore, a contour map which is 512 by 512 will create a terrain 512 units wide and 512 units deep. The greatest height within the terrain will depend on the largest pixel values within the contour image.

We can override the default sizes and specify a multiplying factor for each dimension using the SET TERRAIN SCALE statement which has the format given in FIG-44.14.

FIG-44.14 The SET TERRAIN SCALE Statement



In the diagram:

<i>terno</i>	is an integer value giving the terrain to be scaled.
<i>xscale</i>	is a real number giving the multiplication factor to be applied to the width of the terrain.
<i>yscale</i>	is a real number giving the multiplication factor to be applied to the height of the terrain.

zscale

is a real number giving the multiplication factor to be applied to the depth of the terrain.

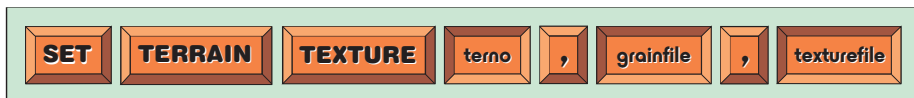
For example, we could double the width, triple the depth and halve the height of terrain object 1 using the statement:

```
SET TERRAIN SCALE 1,2,0.5,3
```

The SET TERRAIN TEXTURE Statement

To texture a terrain we can use the SET TERRAIN TEXTURE statement. This statement allows us to specify two image files. One of these is used to create a grain on the surface of the terrain, while the other supplies the basic texturing image. The SET TERRAIN TEXTURE statement has the format shown in FIG-44.15.

FIG-44.15 The SET TERRAIN TEXTURE Statement



In the diagram:

terno

is an integer value identifying the terrain involved.

grainfile

is a string giving the name of the file to be used to create the underlying surface grain.

texturefile

is a string giving the name of the main texture file.

For example, we could use the file *grain.bmp* to create the grain on terrain 1 and the file *texture.bmp* to texture it using the lines:

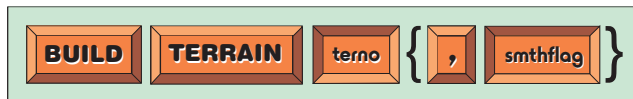
```
LOAD IMAGE "grain.bmp", 1
LOAD IMAGE "texture.bmp", 2
SET TERRAIN TEXTURE 1,1,2
```

The BUILD TERRAIN Statement

Any terrain that we are constructing will become visible only after executing the BUILD TERRAIN statement which has the format shown in FIG-44.16.

FIG-44.16

The BUILD TERRAIN Statement



In the diagram:

terno

is an integer value identifying the terrain involved.

smthflag

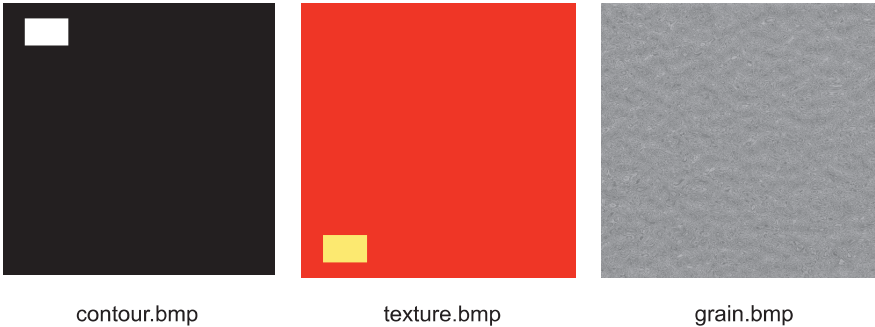
0 or 1. 0 - gives surface smoothing (this is the default); 1 - no smoothing.

When it appears on the screen a terrain object has its bottom-left corner at position (0,0,0).

The program in LISTING-44.3 demonstrates the use of the statements covered so far in this section. The images used are shown in FIG-44.17. Notice that *texture.bmp* is an inverted version of *contour.bmp*. All images are 512 by 512.

FIG-44.17

The Images Used in
Creating the Terrain



LISTING-44.3

Creating a Terrain Using
the Undocumented
Statements

```
#INCLUDE "camera.dba"
SetUpScreen()
REM *** Create terrain object ***
MAKE OBJECT TERRAIN 1
REM *** Assign a contour map ***
SET TERRAIN HEIGHTMAP 1, "contour.bmp"
REM *** Reduce the height ***
SET TERRAIN SCALE 1, 1,0.5,1
REM *** Add grain and texture ***
LOAD IMAGE "grain.bmp",1
LOAD IMAGE "texture.bmp",2
SET TERRAIN TEXTURE 1,1,2
REM *** Show terrain ***
BUILD TERRAIN 1,0
REM *** Allow user to move camera ***
DO
    PointCameraUsingMouse(0)
    MoveCameraUsingMouse(0,1)
LOOP
REM *** End program ***
END

FUNCTION SetUpScreen()
    REM *** Set display mode required ***
    SET DISPLAY MODE 1280,1024,32
    REM *** position camera ***
    AUTOCAM OFF
    POSITION CAMERA 0,256,200,256
    POINT CAMERA 256,0,256
ENDFUNCTION
```

Activity 44.6

Type in and test the program in LISTING-44.3 (*terrain03.dbpro*).

Make sure the *camera.dba* file has been copied to the project folder.

If you look closely, you may notice that the texture file has been tiled over the surface of the terrain. In fact, there's a copy of the image on each square unit of the terrain.

The SET TERRAIN TILING Statement

We can change how often the texture file is tiled over the surface of the terrain using the SET TERRAIN TILING statement which has the format shown in FIG-44.18.

FIG-44.18

The SET TERRAIN
TILING Statement



In the diagram:

<i>terno</i>	is an integer value identifying the terrain involved.
<i>size</i>	is an integer value giving the size of each tile in world units.

For example, if our terrain is 512 units square, then we can force a single copy of the texture image to cover the whole of terrain object 1 using the line:

```
SET TERRAIN TILING 1,512
```

Activity 44.7

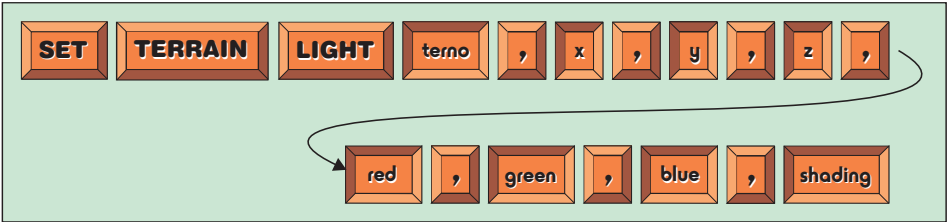
Modify your last program so that a single copy of *texture.bmp* covers the whole of the terrain (the terrain is 512 by 512 units).

Notice that by using an inverted texture file, the contours and textures are an exact match, the yellow in the texture file covering the raised area of the terrain.

The SET TERRAIN LIGHT Statement

We can create a lighting effect on the terrain, controlling the position, colour and intensity of the lights, using the SET TERRAIN LIGHT statement which has the format shown in FIG-44.19.

FIG-44.19 The SET TERRAIN LIGHT Statement



In the diagram:

<i>terno</i>	is an integer value identifying the terrain involved.
<i>x,y,z</i>	are a set of real values giving the position of the light source. No actual light is created, but shading calculations which affect the terrain's surface assume a light is positioned at this point.
<i>red,green,blue</i>	are a set of integer values giving the colour of the light cast on the terrain.
<i>shading</i>	is a real value between 0 and 1 which affects the darkness of shaded areas of the terrain. 0 - darkest shading 1 - lightest shading

Since no light object is created by this statement, other 3D objects in the scene are unaffected by the lighting specified.

We could simulate a blue light shining on terrain 1 from position (10,20,200), creating the darkest shading using the line:

```
SET TERRAIN LIGHT 1,10,20,200,0,0,255,0
```

Activity 44.8

Modify the terrain's lighting in your last program so that dark shading is created for a white light positioned at (-100,10,250).

The SET TERRAIN SPLIT Statement

Normally a terrain is created as a single mesh, but it is possible to split it into a set of meshes using the SET TERRAIN SPLIT statement which has the format shown in FIG-44.20.

FIG-44.20

The SET TERRAIN SPLIT Statement



In the diagram:

- terno* is an integer value identifying the terrain involved.
- value* is an integer value giving the number of splits along each edge of the terrain. Realistically, this will be in the range 2 to 16.

If the split is set to 4, then the terrain will be split into a 4 by 4 set of meshes (that is, 16 meshes). The idea behind this is that as a single mesh the terrain can put a strain on the processor/video card, but as a set of smaller meshes only those meshes that are currently on screen need to be processed.

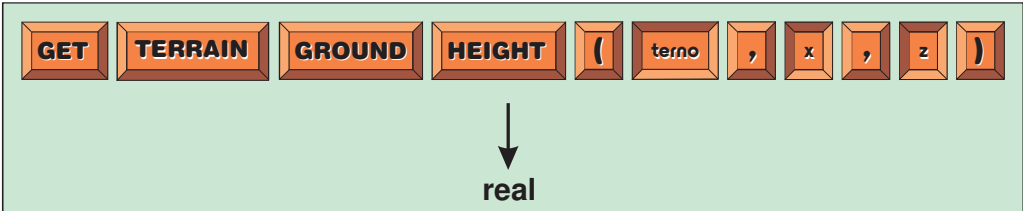
We could split terrain 1 into 64 separate meshes using the statement:

```
SET TERRAIN SPLIT 1, 8
```

The GET TERRAIN GROUND HEIGHT Statement

While we have the GET TERRAIN HEIGHT for old-style terrains, new-style terrains (those created using MAKE OBJECT TERRAIN) use the GET TERRAIN GROUND HEIGHT statement to determine the y-ordinate of a point on the terrain. The statement has the format shown in FIG-44.21.

FIG-44.21 The GET TERRAIN GROUND HEIGHT Statement



In the diagram:

- terno* is the integer value previously assigned to the terrain object.

x,z

are the coordinates of the position within the terrain whose height is to be returned.

The statement returns a real number representing the y-ordinate value at position (x,z) on terrain *terno*. The bottom-left corner of the terrain is taken as position (0,0) irrespective of the terrain object's own position within the 3D world.

The program in LISTING-44.4 repeats the demonstration of moving a cube across a terrain. Notice how much more accurate the demonstration is this time.

LISTING-44.4

Traversing the Terrain

```
SetUpScreen()
REM *** Create a terrain object ***
MAKE OBJECT TERRAIN 1
REM *** Add contour map ***
SET TERRAIN HEIGHTMAP 1,"big2.bmp"
SET TERRAIN SCALE 1,1,0.5,1
REM *** Texture terrain ***
LOAD IMAGE "grain.bmp",1
LOAD IMAGE "grass256.jpg",2
SET TERRAIN TEXTURE 1,1,2
REM *** Makes texture tiles 256 units ***
SET TERRAIN TILING 1, 256
REM *** Show the terrain ***
BUILD TERRAIN 1
REM *** Make and position cube ***
MAKE OBJECT CUBE 2,10
POSITION OBJECT 2,0,0,50
height# = GET TERRAIN GROUND HEIGHT(1,0,50)
REM *** Move cube across terrain ***
FOR c = 0 TO 1023
  REM *** Display current height ***
  SET CURSOR 10,10
  PRINT "CURRENT HEIGHT :",height#
  REM *** Recalculate cube's height and reposition camera ***
  height# = GET TERRAIN GROUND HEIGHT(1,c,50)
  POSITION OBJECT 2,c,height#,50
  POSITION CAMERA c-5,height#+10,0
  POINT CAMERA c,height#,50
  WAIT 1
NEXT c
REM *** End program ***
WAIT KEY
END

FUNCTION SetUpScreen()
  REM *** Set display mode required ***
  SET DISPLAY MODE 1280,1024,32
  COLOR BACKDROP 0
  BACKDROP ON
  REM *** position camera ***
  AUTOCAM OFF
  POSITION CAMERA 0,0,10,-100
  REM *** Increase ambient light ***
  SET AMBIENT LIGHT 50
ENDFUNCTION
```

Activity 44.9

Type in and test the program in LISTING-44.4 (*terrain04.dbpro*).

Before the BUILD TERRAIN statement, add a statement which splits the terrain into 32 by 32 meshes. Does this change have any effect on the speed of the program?

Activity 44.10

Write a program (*act4410.dbpro*) which creates a terrain using the files *contour1.bmp*, *grass256.jpg* and *grass1.bmp*. Use the default tiling setting for the texture. Place the camera at (0,10,0). Set the scaling factors to 1,0.2,1. The terrain should be split into 32 by 32 sections.

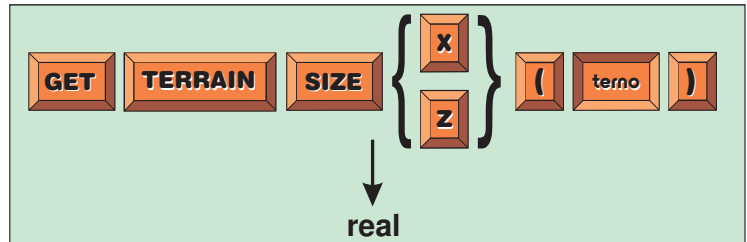
Using GET TERRAIN GROUND HEIGHT, *MoveCameraUsingMouse()* and *PointCameraUsingMouse()* allow the user to move the camera about the terrain keeping a constant 5 units above the surface (remember to copy *camera.dba* to the project file).

The GET TERRAIN SIZE Statement

The dimensions of a terrain object can be obtained using the GET TERRAIN SIZE statement which has the format shown in FIG-44.22.

FIG-44.22

The GET TERRAIN
SIZE Statement



In the diagram:

X,Z

Use the X option to find the width of the terrain.
Use the Z option to find the depth of the terrain.

terno

is an integer value identifying the terrain whose dimensions are to be found.

The function returns a real number representing the specified dimension of the terrain in world units.

For example, to find the width and depth of terrain 1, we would use the lines:

```
width# = GET TERRAIN SIZE X(1)
depth# = GET TERRAIN SIZE Z(1)
```

We could use this information to position the centre of the terrain at the origin:

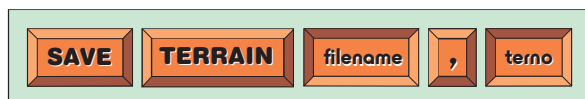
```
POSITION OBJECT 1, width#/2,0,depth#/2
```

The SAVE TERRAIN Statement

Like other 3D objects, a terrain can be saved to a file for use in other programs. This is achieved using the SAVE TERRAIN statement which has the format given in FIG-44.23.

FIG-44.23

The SAVE TERRAIN
Statement



In the diagram:

filename

is a string giving the name of the file into which the terrain is to be saved. The filename should have a *.DBO* extension.

terno

is an integer value identifying the terrain which is to be saved.

For example, we could save terrain 1 to a file named *myterrain.dbo* using the line:

```
SAVE TERRAIN "myterrain.dbo",1
```

Activity 44.11

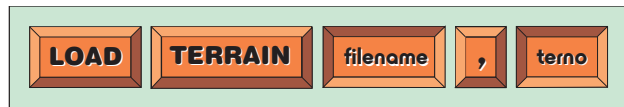
In your last program, add code to save the completed terrain to a file called *myterrain.dbo* when the Enter key is pressed.

The LOAD TERRAIN Statement

A previously saved terrain can be loaded into a program using the LOAD TERRAIN statement which has the format shown in FIG-44.24.

FIG-44.24

The LOAD TERRAIN Statement



In the diagram:

filename

is a string giving the name of the file containing the terrain.

terno

is an integer value specifying the ID to be given to the new terrain object being created.

The loaded terrain will automatically be contoured and textured, assuming the relevant image files are in place. There is no need to call the BUILD TERRAIN statement when loading a terrain from a file.

For example, we can load the previously saved *myterrain.dbo* using the statement:

```
LOAD TERRAIN "myterrain.dbo",1
```

Activity 44.12

Modify your last program to load the terrain used from a file.

In fact, a terrain can be loaded as a regular 3D object with a statement such as

```
LOAD OBJECT "myterrain.dbo",1
```

but it will not be recognised as a terrain and therefore none of the terrain-related statements can be applied to the structure.

Terrains as Objects

In many respects a terrain object can be treated as a regular 3D object. For example, we can use `POSITION OBJECT` to move a terrain to a new position and `ROTATE OBJECT` to rotate it. However, using these statements may affect the accuracy of calls to `GET TERRAIN GROUND HEIGHT`.

If the `TERRAIN SPLIT` statement has been used to split a terrain into several separate meshes, these meshes become limbs and it is possible to hide or delete these limbs.

Activity 44.13

In your last program, add the line

```
HIDE LIMB 1,15
```

before the `DO..LOOP` structure.

What has happened to the terrain?

Other limb commands have an effect - but not always the one you might expect!

Summary

Documented Statements

- Use `MAKE TERRAIN` to create a landscape.
- The contours of the landscape are determined by the image associated with the terrain object.
- The image associated with a terrain object must be square.
- The terrain object's width and depth match that of the image associated with it.
- Use `DELETE TERRAIN` to remove a terrain object from memory.
- When first created, the bottom-left corner of a terrain object is at (0,0,0).
- Place the terrain object within your 3D space using the `POSITION TERRAIN` statement.
- Use `TERRAIN POSITION` to determine the current position of the top-left terrain object.
- Add a texture image to the terrain object using the `TEXTURE TERRAIN` statement.
- The texture image is tiled over the surface of the terrain.
- Determine the height of any point on the terrain object using `GET TERRAIN HEIGHT` statement.

- The GET TOTAL TERRAIN HEIGHT statement returns the y-ordinate value of the highest point on the terrain object.

Undocumented (Advanced Terrain) Statements

- Use MAKE OBJECT TERRAIN to create an advanced terrain object.
- Use SET TERRAIN HEIGHTMAP to specify the image to be used when contouring an advanced terrain.
- Use SET TERRAIN SCALE to resize an advanced terrain in all 3 dimensions.
- Use SET TERRAIN TEXTURE to grain and texture an advanced terrain.
- Use BUILD TERRAIN to make an advanced terrain visible on the screen.
- Use SET TERRAIN TILING to specify the size of each texture tile on the advanced terrain.
- If only one occurrence of the texture image is to be used over the whole terrain, make the tile size equal to the terrain size.
- Use SET TERRAIN LIGHT to specify the position and colour of an assumed light source for the terrain.
- Other objects are not affected by a terrain's light source.
- Use SET TERRAIN SPLIT to convert an advanced terrain to a set of meshes.
- Each mesh within an advanced terrain is treated as a limb of the object.
- Use GET TERRAIN GROUND HEIGHT to discover the height of a specified point within the terrain.
- Use GET TERRAIN SIZE to find the width and depth of a terrain object.
- Use SAVE TERRAIN to save details of a terrain to a file.
- Saved terrains are held in .DBO format.
- Use LOAD TERRAIN to load a previously saved terrain.
- A loaded terrain does not require a call to BUILD TERRAIN.
- A terrain object can be manipulated using traditional object statements such as POSITION OBJECT and ROTATE OBJECT.
- A terrain which has been split into limbs can have those limbs manipulated using standard limb statements.

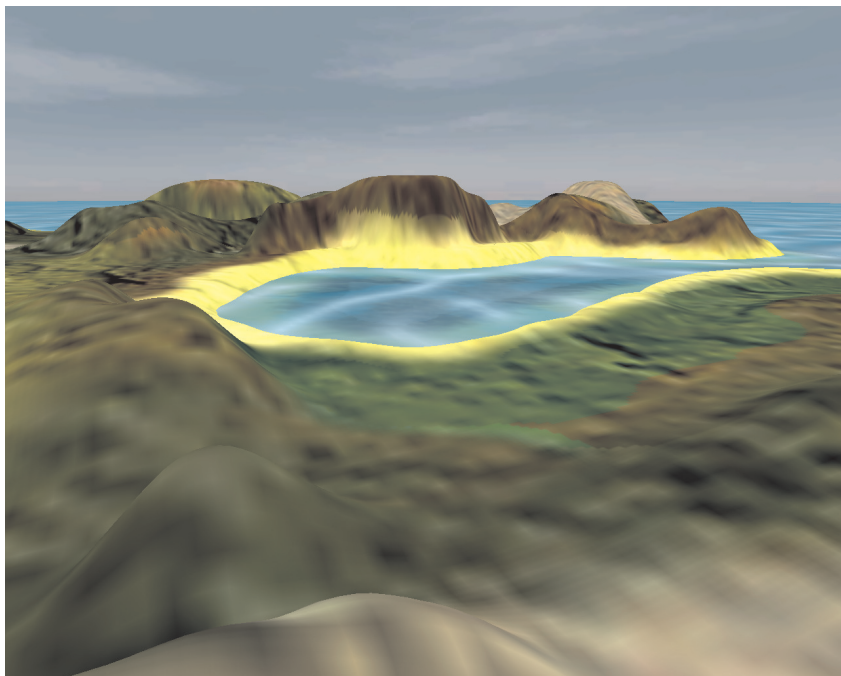
Terrain Project

Introduction

In this extended example we'll see how to use a terrain and a skybox to create the beginnings of a treasure-seeking, first-person perspective game. FIG-44.25 gives a screenshot of the game.

FIG-44.25

The Island



The aim of the game is to wander about the island terrain until you find the golden orb.

Creating the Game

The game logic can, as usual, be described in a few lines:

```
Initialise the game
Show game introduction
REPEAT
    Control player movement
UNTIL golden orb is found
Show end game
```

This logic being coded as:

```
SetUpGame ()
StartGame ()
REPEAT
    ControlPlayer ()
UNTIL treasure reached
EndGame ()
END
```

Constants and Global Variables

The various constants and global variables required are listed below:

```
REM *****
REM ***          Constants          ***
REM *****
REM ***          Booleans          ***
#CONSTANT FALSE          0
#CONSTANT TRUE           1
REM ***          Terrain ID        ***
#CONSTANT IslandTerrain  1
REM ***          Image IDs         ***
#CONSTANT IslandTextureImg 1
#CONSTANT IslandGrainImg   2
#CONSTANT SeaImg           3
#CONSTANT OrbImg           4
#CONSTANT TitleImg         5
#CONSTANT FoundImg         6
REM ***          3D Object IDs     ***
#CONSTANT OrbObj           2
#CONSTANT SkyBoxObj        3
#CONSTANT OceanObj         4
REM ***          Sprites           ***
#CONSTANT TitleSprt        1
#CONSTANT FoundSprt        2

REM *****
REM ***          Globals           ***
REM *****
GLOBAL treasurereached = FALSE      `Indicates if sphere found
GLOBAL spherox#             `Coords of sphere
GLOBAL spherez#
```

ID 1 is already used for the terrain object which is considered to be a normal 3D object.

Activity 44.14

Type in the code (*treasure.dbpro*) given so far and create required Level 1 test stubs. Press Escape to terminate the program.

Adding *StartUpGame()*

Starting up the game involves setting the screen resolution, positioning the camera, and creating the scene, so the logic of this routine can be described as:

```
Set screen resolution
Hide the mouse pointer
Position camera
Create scene
```

Activity 44.15

Write code for the *StartUpGame()* function. **Position camera** and **Create scene** should be implemented in other functions.

The screen resolution should be set to 1280 by 1024 with 32 bit colour.

Add level 2 test stubs for routines named *PositionCamera()* and *CreateScene()*.

Test your program.

Adding *PositionCamera()*

The plan is that the *StartGame()* function will show the camera sweeping in from the sea towards the island, so this routine needs to position the camera out at sea. The code required is:

```
FUNCTION PositionCamera ()
    AUTOCAM OFF
    POSITION CAMERA -800,5,-600
    POINT CAMERA 20,5,100
ENDFUNCTION
```

Activity 44.16

Add the code for the *PositionCamera()* function to your program.

Adding *CreateScene()*

Our game world consists of a terrain-constructed island, a sky, an ocean and the golden orb. Each of these elements will be created by separate routines, so the code for *CreateScene()* will be:

```
FUNCTION CreateScene ()
    LoadTerrain ()
    CreateSkyBox ()
    LoadOcean ()
    PlaceOrb ()
ENDFUNCTION
```

Activity 44.17

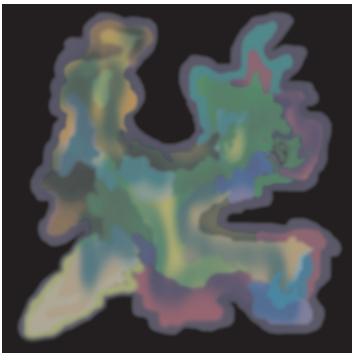
Add the code for the *CreateScene()* function to your program.

Create test stubs for *LoadTerrain()*, *CreateSkyBox()*, *LoadOcean()* and *PlaceOrb()*.

Adding *LoadTerrain()*

The terrain uses the files shown in FIG-44.26. All images are 1024 by 1024.

FIG-44.26 The Images used to Create the Island Terrain



The Contour Map (*islandcontour.bmp*)



The Texture Image (*islandtexture.bmp*)



The Grain Image (*rock.bmp*)

Activity 44.18

Write the code required for *LoadTerrain()*.

In *LoadTerrain()*:

The terrain object should be 1024 by 1024 with a 30% height factor.

Do not allow duplicate tiling when the terrain is textured.

Use POSITION OBJECT to place the centre of the terrain at position (0,0,0).

Use SET OBJECT TRANSPARENCY *IslandTerrain*, 6 to make the black areas of the terrain transparent.

Set the ambient light level to 70.

Adding *CreateSkyBox()*

Back in Chapter 33 we used a sky sphere to create a more realistic scene around our gallows. However, a cube is often used as an alternative to a sphere when creating sky and far-off scenery. We have two options when using a box to create our sky.

One option is to create a model containing 6 plane limbs and texture each separately. We are forced to use planes rather than a cube since it is not possible to texture each side of a cube with different images. Alternatively, we can create a cube in a 3D modelling program and add the textures within that program (where we have more control over what image covers each face of the cube).

Even the images used must be properly prepared if the final effect is to look seamless and natural.

The best option is to use a package designed specifically to create skyboxes.

Luckily, DarkBASIC Pro comes with a skybox in one of its examples, so we'll use that in this program.

The code for the function is:

```
FUNCTION CreateSkyBox()  
    REM *** Load the skybox ***  
    LOAD OBJECT "sb.x", SkyBoxObj  
    REM *** Make it large enough to circle the island ***  
    SCALE OBJECT SkyBoxObj, 20, 20, 20  
    REM *** Tweek the texture so no seam shows ***  
    SET OBJECT TEXTURE SkyBoxObj, 2, 1  
ENDFUNCTION
```

Activity 44.19

Copy the files required by the skybox to the current project folder. The files are: *sb.x*, and *sb1.dds* to *sb6.dds*.

Add the *CreateSkyBox()* function to your program.

Adding *LoadOcean()*

The base of the skybox doesn't look too ocean-like. One way of fixing this is to replace the texture used on the base of the skybox (one of the *sb.dds* files), but another approach is to hide the skybox base with a slightly higher plane and texture

this plane with an appropriate ocean image.

It is this second approach that is used in the code below:

```
FUNCTION LoadOcean()  
    REM *** Create a plane ***  
    MAKE OBJECT PLAIN OceanObj,4096,4096  
    Texture the plane ***  
    LOAD IMAGE "sea.tga",SeaImg  
    TEXTURE OBJECT OceanObj,SeaImg  
    REM *** Position the plane ***  
    XROTATE OBJECT OceanObj,-90  
    POSITION OBJECT OceanObj,0,2,0  
    REM *** Make plane transparent ***  
    SET OBJECT TRANSPARENCY OceanObj,2  
ENDFUNCTION
```

Activity 44.20

Copy *sea.tga* to your project folder.

Add the code for *LoadOcean()* to your project.

Adding *PlaceOrb()*

The last stage of the game initialisation is placing the golden orb somewhere on the island. Since we'd like each game to have a different location for the orb, we'll need to use some random positioning method.

However, we can't choose any position or the orb might end up in the water or buried in a hillside, so we need to check that the position chosen is suitable.

Since the terrain object occupies from point (-512,-512) - bottom-left corner - to (512,512) - top-right corner - as measured on the XZ plane, we need to generate an x-ordinate in the range -512 to +512. The same range is required for the z-ordinate. This can be achieved using the lines:

```
x# = RND(1024)-512  
z# = RND(1024)-512
```

To discover if we have chosen a suitable point on the island, all we need to do is check the height of the terrain at that point. If its less than say 5, then we are in the water or by the coast and so should choose another point.

From this we can come up with an outline logic for the *PlaceOrb()* function:

```
Seed the random number generator  
REPEAT  
    Generate random x-ordinate  
    Generate random z-ordinate  
    Set y-ordinate to ground height at position (x,z)  
UNTIL y-ordinate > 5  
Create orb object  
Position orb at point (x, y,z)
```

Activity 44.21

From the logic given above, add the code for *PlaceOrb()* to your project.

Texture orb using
gold.bmp

Adding *StartGame()*

The *StartGame()* function is going to fly the camera in over the ocean towards the island while showing the game title at the top of the screen.

The game title is held as a sprite (loaded from file *title.bmp*).

This routine requires the following code:

```
FUNCTION StartGame()  
    REM *** Show title  
    LOAD IMAGE "title.bmp",TitleImg  
    SPRITE TitleSprt,400,100,TitleImg  
    REM *** Move camera to shore ***  
    FOR c = 1 TO 330  
        MOVE CAMERA 2  
        WAIT 1  
    NEXT c  
    WAIT 1000  
    REM *** Hide title ***  
    HIDE SPRITE TitleSprt  
    DELETE SPRITE TitleSprt  
ENDFUNCTION
```

Activity 44.22

Add the *StartGame()* code to your program.

Adding *ControlPlayer()*

This routine will allow the player to take control of the camera using the mouse. As a first attempt we need only use the *PointCameraUsingMouse()* and *MoveCameraUsingMouse()* routines that we developed in a previous chapter. However, we must adjust the height of the camera to be slightly above the terrain it is travelling along.

The code required is:

```
FUNCTION ControlPlayer()  
    PointCameraUsingMouse(0)  
    MoveCameraUsingMouse(0,1)  
    REM *** Lift camera above ground height ***  
    x# = CAMERA POSITION X(0)  
    z# = CAMERA POSITION Z(0)  
    y# = GET TERRAIN GROUND HEIGHT(IslandTerrain,x#,z#)+5  
    POSITION CAMERA 0,x#,y#,z#  
ENDFUNCTION
```

Activity 44.23

Add the code for *ControlPlayer()* to your program.

The trouble with this simple version of *ControlPlayer()* is that the player can walk on water and climb the steepest hills with ease. A more realistic version of the routine would limit the the player to the land and stop any steep climbs or descents. To achieve this, all we need to do is check the camera's new position after each move and if it is too low (in the water) or is too large a change in the y-ordinate (ascending/descending steep cliffs), then we can move the camera back to its earlier position.

Assuming *y#* contains the camera's current y-ordinate and *oldy#* contains its previous y-ordinate, then we can restrict movement by adding the following lines:

```
REM *** IF too low or large change in height ***
IF (y# < 7.5) OR (ABS(oldy#-y#)>1)
    REM *** Return camera to previous position ***
    POSITION CAMERA 0, oldx#,oldy#,oldz#
ENDIF
```

The code for the new version of the routine is :

```
FUNCTION ControlPlayer()
    REM *** Get camera's current position ***
    oldx# = CAMERA POSITION X(0)
    oldy# = CAMERA POSITION Y(0)
    oldz# = CAMERA POSITION Z(0)
    REM *** Move camera ***
    PointCameraUsingMouse(0)
    MoveCameraUsingMouse(0,1)
    REM *** Lift camera above ground height ***
    x# = CAMERA POSITION X(0)
    z# = CAMERA POSITION Z(0)
    y# = GET TERRAIN GROUND HEIGHT(IslandTerrain,x#,z#)+5
    POSITION CAMERA 0, x#,y#,z#
    REM *** IF too low or large change in height ***
    IF (y# < 7.5) OR (ABS(oldy#-y#)>1)
        REM *** Return camera to previous position ***
        POSITION CAMERA 0, oldx#,oldy#,oldz#
    ENDIF
ENDFUNCTION
```

But even now, the routine is not quite finished. It also needs to detect when the camera is close to the sphere so that the global variable, *treasurefound*, can be set to true. This is done by ending the routine with the lines:

```
REM *** IF sphere reached, set variable ***
IF CameraBesideOrb()
    treasurereached = TRUE
ENDIF
```

Activity 44.24

Add the code for *ControlPlayer()* to your program.

Write the code for *CameraBesideOrb()* which should return 1 if the x and z ordinates of the camera and sphere are both within 5 units of each other; otherwise zero is returned.

Adding *EndGame()*

This function will flash up the message FOUND for 3 seconds before the program shuts down. The message is created as an animated sprite (see FIG-44.27).

FIG-44.27

The Animated Sprite
Message



The code for this routine is:

```
FUNCTION EndGame()
    REM *** Load sprite ***
```

```

CREATE ANIMATED SPRITE FoundSprt, "found.bmp", 2, 1, FoundImg
SPRITE FoundSprt, 400, 500, FoundImg
REM *** Show it for 3 seconds ***
now = TIMER()
WHILE TIMER() - now < 3000
    PLAY SPRITE FoundSprt, 1, 2, 20
ENDWHILE
ENDFUNCTION

```

Activity 44.25

Add this last routine to your program and test the completed project.

Adding Testing Features

It seems to be a popular pastime to include cheats in games. If you know what keys to press you can give yourself unlimited lives, increase your speed, gain greater firepower, and many other options.

These features are not always included just to help out the keen player; a programmer will often add features to a game to help with the testing. For example, testing the island game can be time-consuming if you have no idea where the golden orb is - or where you are, for that matter.

It would be useful if we knew the coordinates of both the sphere and the camera. That way, we could find the orb quickly and make sure that the game ends correctly.

In the code that follows, we'll let the player find out the position of the orb by pressing the "s" key and the position of the camera by hitting "c". To do this we'll need to change the coding of the main section to read:

```

SetUpGame()
StartGame()
REPEAT
    ControlPlayer()
    IF INKEY$() = "s"
        res$ = STR$(spherex#) + " " + STR$(spherez#)
        DisplayMessage(res$, 200, 200)
    ENDIF
    IF INKEY$() = "c"
        res$ = STR$(CAMERA POSITION X(0)) + " "
        + STR$(CAMERA POSITION Z(0))
        DisplayMessage(res$, 200, 200)
    ENDIF
UNTIL treasurereached
EndGame()
END

```

The new function, *DisplayMessage()*, is one we've used before and is coded as:

```

REM *****
REM ***      Debug Routines      ***
REM *****
FUNCTION DisplayMessage(mes$, x, y)
    now = TIMER()
    WHILE TIMER() - now < 1000
        SET CURSOR x, y
        PRINT mes$
    ENDWHILE
ENDFUNCTION

```

Notice the function is placed in a special section for debugging (testing) routines.

Activity 44.26

Update your project to reflect these changes and use the secret keys to help locate the orb.

Code added specifically to aid debugging would often be removed from the final product after testing is complete, but sometimes left in to allow those in the know to gain an advantage over others (cheat!).

Solutions

Activity 44.1

To use *terrain02.bmp*, the main section changes to:

```
SetUpScreen()
REM *** Create terrain ***
MAKE TERRAIN 1,"terrain02.bmp"
REM *** Allow user to move camera ***
DO
    CONTROL CAMERA USING ARROWKEYS 0,5,1
LOOP
REM *** End program ***
WAIT KEY
END
```

When using your own image the highlighted line would need to be changed to contain the name of your file.

Activity 44.2

Change the main section of the program to:

```
#INCLUDE "camera.dba"
SetUpScreen()
REM *** Create terrain ***
MAKE TERRAIN 1,"terrain02.bmp"
REM *** Reposition centre to (0,0,0)***
REM *** Find size of terrain ***
LOAD BITMAP "terrain02.bmp",1
size = BITMAP WIDTH(1)
DELETE BITMAP 1
POSITION TERRAIN 1, -size/2,0,-size/2
REM *** Move camera to centre ***
POSITION CAMERA 0,0,50,0
REM *** Allow user to move camera ***
DO
    PointCameraUsingMouse(0)
    MoveCameraUsingMouse(0,1)
LOOP
REM *** End program ***
END
```

Activity 44.3

```
PRINT TERRAIN POSITION Y(2)
```

Activity 44.4

The main section should be changed to:

```
#INCLUDE "camera.dba"
SetUpScreen()
REM *** Create terrain ***
MAKE TERRAIN 1,"terrain02.bmp"
REM *** Reposition terrain to (0,0,0)
POSITION TERRAIN 1, 0, 0, 0
REM *** Find size of terrain ***
LOAD BITMAP "terrain03.jpg",1
size = BITMAP WIDTH(1)
DELETE BITMAP 1
REM *** Move camera to centre ***
POSITION CAMERA 0, size/2,50,-size/2
REM *** Texture terrain ***
LOAD IMAGE "grass256.jpg",2
TEXTURE TERRAIN 1,2
REM *** Allow user to move camera ***
DO
    PointCameraUsingMouse(0)
```

```
MoveCameraUsingMouse(0,1)
LOOP
REM *** End program ***
WAIT KEY
END
```

The camera can pass straight through the terrain.

Activity 44.5

The height value returned by GET TERRAIN HEIGHT correctly returns the value of zero when the terrain is at its lowest level, but returns a value of 100 for the highest level when, in fact, its height in units is only 65.

Activity 44.6

No solution required.

Activity 44.7

The main section should be changed as shown below:

```
#INCLUDE "camera.dba"
SetUpScreen()
REM *** Create terrain object ***
MAKE OBJECT TERRAIN 1
REM *** Assign a contour map ***
SET TERRAIN HEIGHTMAP 1, "contour.bmp"
REM *** Reduce the height ***
SET TERRAIN SCALE 1, 1,0.5,1
REM *** Add grain and texture ***|
LOAD IMAGE "grain.bmp",1
LOAD IMAGE "texture.bmp",2
SET TERRAIN TEXTURE 1,1,2
REM *** No tiling ***
SET TERRAIN TILING 1,512
REM *** Show terrain ***
BUILD TERRAIN 1,0
REM *** Allow user to move camera ***
DO
    PointCameraUsingMouse(0)
    MoveCameraUsingMouse(0,1)
LOOP
REM *** End program ***
END
```

Activity 44.8

To add the light effect add the following lines to the main section before the BUILD TERRAIN statement:

```
REM *** Set up terrain lighting ***
SET TERRAIN LIGHT 1, -100,10,250,0
```

Activity 44.9

The statement required to split the terrain into the required number of tiles is:

```
SET TERRAIN SPLIT 1, 32
```

The smoothness of the camera movement should improve.

Activity 44.10

```
#INCLUDE "camera.dba"
SetUpScreen()
REM *** Create a terrain object ***
MAKE OBJECT TERRAIN 1
REM *** Add contour map ***
SET TERRAIN HEIGHTMAP 1,"contour1.bmp"
SET TERRAIN SCALE 1,1,0.2,1
REM *** Texture terrain ***
LOAD IMAGE "grain.bmp",1
LOAD IMAGE "grass256.jpg",2
SET TERRAIN TEXTURE 1,1,2
REM *** Partition terrain ***
SET TERRAIN SPLIT 1,32
REM *** Show the terrain ***
BUILD TERRAIN 1
REM *** Allow the user to move camera ***
DO
    PointCameraUsingMouse(0)
    MoveCameraUsingMouse(0,1)
    x# = CAMERA POSITION X(0)
    z# = CAMERA POSITION Z(0)
    y# = GET TERRAIN GROUND HEIGHT(1,x#,z#)
    POSITION CAMERA 0,x#,y#+5,z#
LOOP
REM *** End program ***
END

FUNCTION SetUpScreen()
    REM *** Set display mode required ***
    SET DISPLAY MODE 1280,1024,32
    COLOR BACKDROP 0
    BACKDROP ON
    REM *** position camera ***
    AUTOCAM OFF
    POSITION CAMERA 0,0,10,0
    REM *** Increase ambient light ***
    SET AMBIENT LIGHT 50
ENDFUNCTION
```

Activity 44.11

Change the DO..LOOP in the main section to read:

```
DO
    PointCameraUsingMouse(0)
    MoveCameraUsingMouse(0,1)
    x# = CAMERA POSITION X(0)
    z# = CAMERA POSITION Z(0)
    y# = GET TERRAIN GROUND HEIGHT(1,x#,z#)
    POSITION CAMERA 0, x#,y#+5,z#
    REM *** Save to file on ENTER ***
    IF INKEY$( ) = CHR$(13)
        SAVE TERRAIN "myterrain.dbo",1
    ENDIF
LOOP
```

Activity 44.12

The main section should be changed to:

```
#INCLUDE "camera.dba"
SetUpScreen()
REM *** Load terrain from file ***
LOAD TERRAIN "myterrain.dbo",1
REM *** Allow user to move camera ***
DO
    PointCameraUsingMouse(0)
    MoveCameraUsingMouse(0,1)
    x# = CAMERA POSITION X(0)
    z# = CAMERA POSITION Z(0)
    y# = GET TERRAIN GROUND HEIGHT(1,x#,z#)
    POSITION CAMERA 0, x#,y#+5,z#
LOOP
```

```
REM *** End program ***
END
```

Activity 44.13

Adding the HIDE LIMB statement will cause one section of the terrain to disappear.

Activity 44.14

This initial core of the program is coded as:

```
REM *****
REM *** Constants ***
REM *****
REM *** Booleans ***
#CONSTANT FALSE 0
#CONSTANT TRUE 1
REM *** Terrain ID ***
#CONSTANT IslandTerrain 1
REM *** Image IDs ***
#CONSTANT IslandTextureImg 1
#CONSTANT IslandGrainImg 2
#CONSTANT SeaImg 3
#CONSTANT OrbImg 4
#CONSTANT TitleImg 5
#CONSTANT FoundImg 6
REM *** 3D Object IDs ***
#CONSTANT OrbObj 2
#CONSTANT SkyBoxObj 3
#CONSTANT OceanObj 4
REM *** Sprites ***
#CONSTANT TitleSprt 1
#CONSTANT FoundSprt 2
REM *****
REM *** Globals ***
REM *****
GLOBAL treasureReached = FALSE
    ↳ Indicates if sphere found
GLOBAL sphereX#
    ↳ Coords of sphere
GLOBAL sphereZ#
```

```
REM *** Main logic ***
SetUpGame()
StartGame()
REPEAT
    ControlPlayer()
UNTIL treasureReached
EndGame()
END
```

```
REM *****
REM *** Level 1 ***
REM *****
```

```
FUNCTION SetUpGame()
ENDFUNCTION
```

```
FUNCTION StartGame()
ENDFUNCTION
```

```
FUNCTION ControlPlayer()
ENDFUNCTION
```

```
FUNCTION EndGame()
ENDFUNCTION
```

Activity 44.15

The code for *SetUpGame()* should be:

```

FUNCTION SetUpGame ()
    SET DISPLAY MODE 1280,1024,32
    HIDE MOUSE
    PositionCamera ()
    CreateScene ()
ENDFUNCTION

```

The additional test stubs should be added in a new Level 2 section of the code:

```

REM *****
REM ***           Level 2           ***
REM *****

```

```

FUNCTION PositionCamera ()
ENDFUNCTION

```

```

FUNCTION CreateScene ()
ENDFUNCTION

```

Activity 44.16

The code given should replace the existing level 2 test stub for *PositionCamera()*.

Activity 44.17

Level 2 and Level 3 code should now be:

```

REM *****
REM ***           Level 2           ***
REM *****
FUNCTION PositionCamera ()
    AUTOCAM OFF
    POSITION CAMERA -800,8,-600
    POINT CAMERA 20,8,100
ENDFUNCTION

```

```

FUNCTION CreateScene ()
    LoadTerrain ()
    CreateSkyBox ()
    LoadOcean ()
    PlaceOrb ()
ENDFUNCTION

```

```

REM *****
REM ***           Level 3           ***
REM *****
FUNCTION LoadTerrain ()
ENDFUNCTION

```

```

FUNCTION CreateSkyBox ()
ENDFUNCTION

```

```

FUNCTION LoadOcean ()
ENDFUNCTION

```

```

FUNCTION PlaceOrb ()
ENDFUNCTION

```

Activity 44.18

The code for *LoadTerrain()* is:

```

FUNCTION LoadTerrain ()
    REM *** Create terrain (1024 by 1024) ***
    MAKE OBJECT TERRAIN IslandTerrain
    SET TERRAIN HEIGHTMAP IslandTerrain,
    ↵ "islandcontour.bmp"

    SET TERRAIN SCALE IslandTerrain,1,0.3,1
    LOAD IMAGE "islandtexture.bmp",
    ↵ IslandTextureImg
    LOAD IMAGE "rock.bmp", IslandGrainImg

```

```

SET TERRAIN TEXTURE IslandTerrain,
↵ IslandGrainImg, IslandTextureImg
SET TERRAIN TILING IslandTerrain,1024
SET TERRAIN SPLIT IslandTerrain,32
BUILD TERRAIN IslandTerrain
REM *** Place centre at (0,0,0) ***
POSITION OBJECT 1, -512,0,-512
REM *** Make black transparent ***
SET OBJECT TRANSPARENCY IslandTerrain,6
REM *** Increase ambient light ***
SET AMBIENT LIGHT 70
ENDFUNCTION

```

Activity 44.19

A sky should now appear around the island.

Activity 44.20

A semi-transparent sea should now appear within the scene.

Activity 44.21

The code for *PlaceOrb()* is:

```

FUNCTION PlaceOrb ()
    REM *** Seed random number generator ***
    RANDOMIZE TIMER ()
    REM *** Generate a valid position ***
    REPEAT
        x# = RND (1024)-512
        z# = RND (1024)-512
        y# = GET TERRAIN GROUND HEIGHT
        ↵ (IslandTerrain,x#,z#)
    UNTIL y# > 5
    REM *** Create and place sphere ***
    MAKE OBJECT SPHERE OrbObj,2
    LOAD IMAGE "gold.bmp",OrbImg
    TEXTURE OBJECT OrbObj,OrbImg
    POSITION OBJECT OrbObj,x#,y#+1,z#
    REM *** Record its position ***
    spherex# = x#
    spherez# = z#
ENDFUNCTION

```

Activity 44.22

The code should replace the Level 1 test stub for *StartGame()*.

Activity 44.23

The code should replace the Level 1 test stub for *ControlPlayer()*.

Remember to copy the *camera.dba* file to the project folder and to add the line

```
#INCLUDE "camera.dba"
```

at the start of the program.

Activity 44.24

The code for *CameraBesideOrb()* is:

```

FUNCTION CameraBesideOrb ()
    IF ABS (CAMERA POSITION X (0)-spherex#) <5
    ↵ AND ABS (CAMERA POSITION Z (0)-spherez#)

```

```

    <5
    result = 1
ELSE
    result = 0
ENDIF
ENDFUNCTION result

```

Activity 44.25

No solution required.

Activity 44.26

The complete program (including debugging code) is:

```

#INCLUDE "camera.dba"
REM *****
REM *** Constants ***
REM *****
REM *** Booleans ***
#CONSTANT FALSE 0
#CONSTANT TRUE 1
REM *** Terrain ID ***
#CONSTANT IslandTerrain 1
REM *** Image IDs ***
#CONSTANT IslandTextureImg 1
#CONSTANT IslandGrainImg 2
#CONSTANT SeaImg 3
#CONSTANT OrbImg 4
#CONSTANT TitleImg 5
#CONSTANT FoundImg 6
REM *** 3D Object IDs ***
#CONSTANT OrbObj 2
#CONSTANT SkyBoxObj 3
#CONSTANT OceanObj 4
REM *** Sprites ***
#CONSTANT TitleSprt 1
#CONSTANT FoundSprt 2
REM *****
REM *** Globals ***
REM *****
`Indicates if sphere found
GLOBAL treasurereached = FALSE
`Coords of sphere
GLOBAL spherex#
GLOBAL spherez#

REM *** Main logic ***
SetUpGame()
StartGame()
REPEAT
    ControlPlayer()
    IF INKEY$()="s"
        res$ = STR$(spherex#)+" "
        <+STR$(spherez#)
        DisplayMessage(res$,200,200)
    ENDIF
    IF INKEY$()="c"
        res$ = STR$(CAMERA POSITION X(0))+
        <" "+STR$(CAMERA POSITION Z(0))
        DisplayMessage(res$,200,200)
    ENDIF
UNTIL treasurereached
EndGame()
END

REM *****
REM *** Level 1 ***
REM *****
FUNCTION SetUpGame()
    SET DISPLAY MODE 1280,1024,32
    HIDE MOUSE
    PositionCamera()
    CreateScene()
ENDFUNCTION

```

```

FUNCTION StartGame()
    REM *** Show title
    LOAD IMAGE "title.bmp",TitleImg
    SPRITE TitleSprt,400,100,TitleImg
    REM *** Move camera to shore ***
    FOR c = 1 TO 333
        MOVE CAMERA 2
        WAIT 1
    NEXT c
    WAIT 1000
    REM *** Hide title ***
    HIDE SPRITE TitleSprt
    DELETE SPRITE TitleSprt
ENDFUNCTION

FUNCTION ControlPlayer()
    REM *** Get camera's position ***
    oldx# = CAMERA POSITION X(0)
    oldy# = CAMERA POSITION Y(0)
    oldz# = CAMERA POSITION Z(0)
    REM *** Move camera ***
    PointCameraUsingMouse(0)
    MoveCameraUsingMouse(0,1)
    REM *** Lift camera above ground ***
    x# = CAMERA POSITION X(0)
    z# = CAMERA POSITION Z(0)
    y# = GET TERRAIN GROUND HEIGHT
    <(IslandTerrain,x#,z#)+5
    POSITION CAMERA 0, x#,y#,z#
    REM *** IF too low or large change ***
    IF (y# < 7.5) OR (ABS(oldy#-y#)>1)
        REM *** Return to old position ***
        POSITION CAMERA 0, oldx#,oldy#,oldz#
    ENDIF
    REM *** If orb found, set indicator ***
    IF CameraBesideOrb()
        treasurereached = TRUE
    ENDIF
ENDFUNCTION

FUNCTION EndGame()
    CREATE ANIMATED SPRITE FoundSprt,
    <"found.bmp",2,1,FoundImg
    SPRITE FoundSprt,400,500,FoundImg
    now = TIMER()
    WHILE TIMER() - now<3000
        PLAY SPRITE FoundSprt,1,2,20
    ENDWHILE
ENDFUNCTION

REM *****
REM *** Level 2 ***
REM *****
FUNCTION PositionCamera()
    AUTOCAM OFF
    POSITION CAMERA -800,8,-600
    POINT CAMERA 20,8,100
ENDFUNCTION

FUNCTION CreateScene()
    LoadTerrain()
    CreateSkyBox()
    LoadOcean()
    PlaceOrb()
ENDFUNCTION

REM *****
REM *** Level 3 ***
REM *****
FUNCTION LoadTerrain()
    REM *** Create terrain (1024 by 1024)***
    MAKE OBJECT TERRAIN IslandTerrain
    SET TERRAIN HEIGHTMAP IslandTerrain,
    <"islandcontour.bmp"
    SET TERRAIN SCALE IslandTerrain,1,0.3,1
    LOAD IMAGE "islandtexture.bmp",
    <IslandTextureImg

```

```

LOAD IMAGE "rock.bmp",IslandGrainImg
SET TERRAIN TEXTURE IslandTerrain,
↳IslandGrainImg,IslandTextureImg
SET TERRAIN TILING IslandTerrain,1024
SET TERRAIN SPLIT IslandTerrain,32
BUILD TERRAIN IslandTerrain
REM *** Position centre at (0,0,0) ***
POSITION OBJECT 1, -512,0,-512
REM *** Black areas (sea)transparent ***
SET OBJECT TRANSPARENCY IslandTerrain,6
REM *** Increase ambient light ***
SET AMBIENT LIGHT 70
ENDFUNCTION

```

```

FUNCTION CreateSkyBox()
LOAD OBJECT "sb.x",SkyBoxObj
SCALE OBJECT SkyBoxObj,20,20,20
SET OBJECT TEXTURE SkyBoxObj,2,1
ENDFUNCTION

```

```

FUNCTION LoadOcean()
MAKE OBJECT PLAIN OceanObj,4096,4096
LOAD IMAGE "sea.tga",SeaImg
TEXTURE OBJECT OceanObj,SeaImg
XROTATE OBJECT OceanObj,-90
POSITION OBJECT OceanObj,0,2,0
SET OBJECT TRANSPARENCY OceanObj,2
ENDFUNCTION

```

```

FUNCTION PlaceOrb()
REM *** Seed random number generator ***
RANDOMIZE TIMER()
REM *** Generate a valid position ***
REPEAT
x# = RND(1024)-512
z# = RND(1024)-512
y# = GET TERRAIN GROUND HEIGHT
↳(IslandTerrain,x#,z#)
UNTIL y# > 5
REM *** Create and place sphere ***
MAKE OBJECT SPHERE OrbObj,2
LOAD IMAGE "gold.bmp",OrbImg
TEXTURE OBJECT OrbObj,OrbImg
POSITION OBJECT OrbObj,x#,y#+1,z#
REM *** Record its position ***
spherex# = x#
spherez# = z#
ENDFUNCTION

```

```

FUNCTION CameraBesideOrb()
IF ABS(CAMERA POSITION X(0)-spherex#)<5
↳AND ABS(CAMERA POSITION Z(0)-spherez#)
↳<5
result = 1
ELSE
result = 0
ENDIF
ENDFUNCTION result

```

```

REM *****
REM *** Debug Routines ***
REM *****
FUNCTION DisplayMessage(mes$,x,y)
now = TIMER()
WHILE TIMER() - now < 1000
SET CURSOR x,y
PRINT mes$
ENDWHILE
ENDFUNCTION

```

How to Use a Matrix Object to Create a Landscape

Creating a Matrix Object

Assigning Random Heights to a Matrix

Assigning Specific Heights to a Matrix

Texturing a Matrix Object

Creating Image Tiles

Setting Matrix Transparency

Setting Matrix Drawing Priority

Scrolling the Matrix Texture

Determining the Height of a Point on the Matrix

Positioning the Matrix

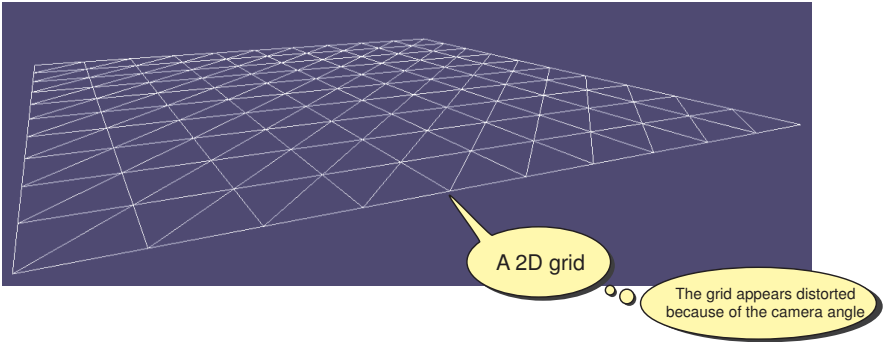
Introduction

Although we can create ground surface using the terrain objects as described in the previous chapter, an alternative way to create an undulating surface is to use a **matrix** object.

DarkBASIC Pro uses the term matrix to describe a two-dimensional plane which is divided into a number of rows and columns, collectively known as **subdivisions**. This creates a grid pattern in the plane (see FIG-45.1).

FIG-45.1

A Flat Matrix

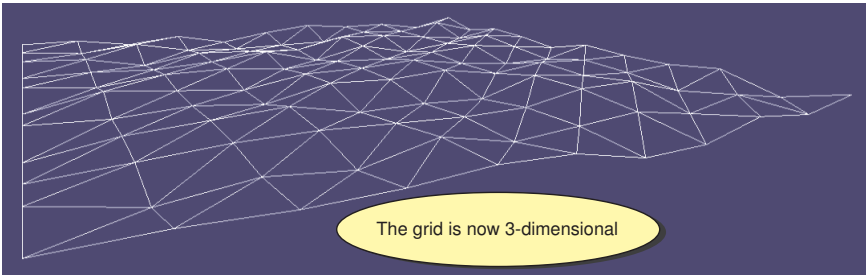


The intersecting subdivisions create a set of rectangular areas and each rectangle is divided into two triangles.

The y-ordinate of individual apexes which make up the triangles within the structure can be manipulated to create a 3D effect (see FIG-45.2).

FIG-45.2

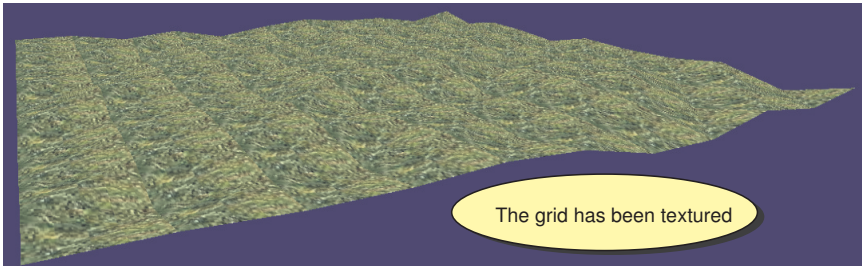
Adding a Third Dimension to the Matrix



A matrix, like any other 3D object, can also have a texture applied to it (see FIG-45.3) allowing you to create a landscape effect similar to that produced by a terrain object.

FIG-45.3

Texturing a Matrix



The set of commands available to create and manipulate a matrix are described in the next section.

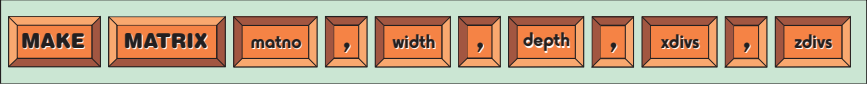
Creating a Matrix

The MAKE MATRIX Statement

We start by creating a flat surface lying horizontally in the XZ plane using the MAKE MATRIX statement. This statement allows us to specify the dimensions of the matrix grid and the number of rows and columns in the grid (that is, the number of subdivisions along the x and z axes). The statement has the format shown in FIG-45.4.

FIG-45.4

The MAKE MATRIX Statement



In the diagram:

<i>matno</i>	is an integer value to be assigned to the matrix. No two matrices may have the same ID.
<i>width</i>	is a real number giving the width of the surface along the x-axis.
<i>depth</i>	is a real number giving the depth of the plane along the z-axis.
<i>xdivs</i>	is an integer value specifying how many subdivisions are required along the x-axis.
<i>zdivs</i>	is an integer value specifying how many subdivisions are required along the z-axis.

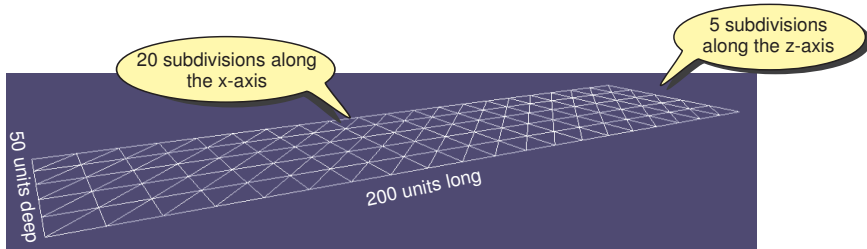
For example, the line

```
MAKE MATRIX 1,200,50,20,5
```

will create the grid shown in FIG-45.5.

FIG-45.5

A Typical Matrix



Activity 45.1

Write a statement to create a matrix grid (with an ID of 2) that is 1000 units deep by 500 units wide.

Make each tile 100 by 100 units.

The grid is initially positioned so that its bottom-left corner is at the origin (0,0,0).

The RANDOMIZE MATRIX Statement

The quickest way to add a third dimension to a matrix grid is to use the RANDOMIZE MATRIX statement which randomly assigns a value (within a specified limit) to the y-ordinate of the four corners of every tile in the grid. This statement has the format shown in FIG-45.6.

FIG-45.6

The RANDOMIZE
MATRIX Statement



In the diagram:

matno is an integer value giving the matrix ID.

ymax is a real value representing the maximum positive displacement of each corner along the y-axis.

By executing this statement, every corner in the grid will be displaced by a random amount between 0 and *ymax*. For example, assuming we began by creating a four tile matrix grid with the statement

MAKE MATRIX 1,100,100,2,2

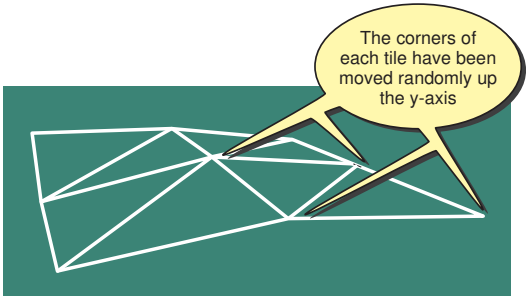
we could randomly reposition the corners of each tile by a value of between 0 and 10.5 using the statement:

RANDOMIZE MATRIX 1,10.5

The resulting grid will, of course, vary each time the program is run, but a typical example is shown in FIG-45.7.

FIG-45.7

The Effect of the
RANDOMIZE MATRIX
Statement



The UPDATE MATRIX Statement

If we make any change to a matrix grid (such as executing the RANDOMIZE MATRIX command), that change won't show up on the screen until the program executes the UPDATE MATRIX statement which has the format shown in FIG-45.8.

FIG-45.8

The UPDATE MATRIX
Statement



In the diagram:

matno is an integer value giving the matrix ID.

For example, to show any changes to grid 1 we would use the line:

We now have enough knowledge to create our first 3D matrix grid as shown in LISTING-45.1.

LISTING-45.1

Creating a Matrix Grid

```

SetUpScreen()
REM *** Create a grid (800 by 720) ***
MAKE MATRIX 1,800,720,8,9
REM *** Wait for key press ***
WAIT KEY
REM *** Randomly adjust the y ordinates ***
REM *** to a maximum of 40 ***
RANDOMIZE MATRIX 1,40
REM *** Update matrix ***
UPDATE MATRIX 1
REM *** End program ***
WAIT KEY
END

FUNCTION SetUpScreen()
  SET DISPLAY MODE 1280,1024,32
  REM *** Position camera for a good view of matrix ***
  AUTOCAM OFF
  POSITION CAMERA 0, 500,300,-600
ENDFUNCTION

```

Activity 45.2

Type in and test the program given in LISTING-45.1 (*matrix01.dbpro*).

Modify the grid to be 1000 by 500 with each rectangle being 10 by 10.

Set the random height variation to range from 0 to 20. Run the program to check the effect of these changes.

Activity 45.3

Modify your program again so that the camera position can be altered using the arrow keys. Start with the camera at position (0,100,-300).

Move the camera over the grid to check that the changes are operating correctly.

The SET MATRIX HEIGHT Statement

If we want more control over the positioning of tile corners, we can use the SET MATRIX HEIGHT statement which allows us to give an exact amount of movement to a specific tile corner. This statement's format is shown in FIG-45.9.

FIG-45.9 The SET MATRIX HEIGHT Statement



In the diagram:

matno

is an integer value giving the matrix ID.

xedge

is an integer value specifying an edge along the x-axis. The left-most edge has the value zero.

zedge

is an integer value specifying an edge along the z-axis. The closest edge has the value zero.

height

is a real value representing the height to which the specified corner is to be raised above the plane of the grid. This can be a positive or negative value.

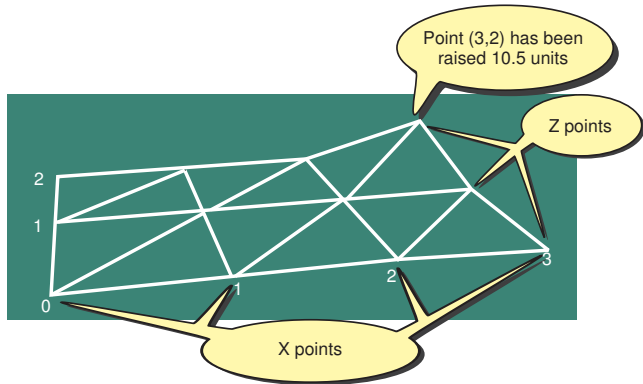
The result of the following statements

```
MAKE MATRIX 1,150,100,3,2
SET MATRIX HEIGHT 1,3,2,10.5
UPDATE MATRIX 1
```

is shown in FIG-45.10.

FIG-45.10

Using the SET MATRIX
HEIGHT Statement



Activity 45.4

Remove the RANDOMIZE MATRIX statement from your last program.

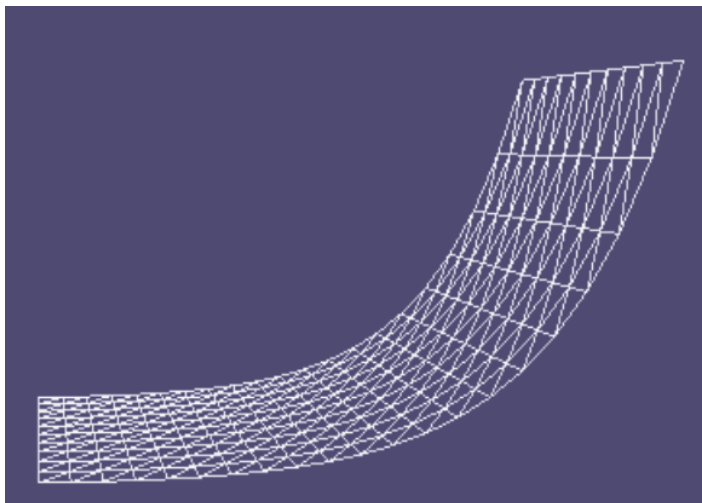
Add two new statements to the program which:

- lifts the corner (8,3) 25.7 units.
- drops corner (0,1) -12.5 units

By using nested FOR loops we can assign specific values to every point in the grid. For example, the program in LISTING-45.2 creates a rising ramp by increasing the size of every point as we move from the left to the right of a grid (see FIG-45.11).

FIG-45.11

A Controlled Matrix



LISTING-45.2

Creating a Structured Grid

```
SetUpScreen()  
  
REM *** Set up 200 by 100 grid (20 by 10 tiles) ***  
MAKE MATRIX 1,200,100,20,10  
REM *** Set initial height ***  
height#=0.5  
REM *** FOR each x edge DO ***  
FOR x = 0 TO 20  
    REM *** Increase height by 30% ***  
    height# = height# *1.3  
    REM *** FOR each z edge DO ***  
    FOR z = 0 TO 10  
        REM *** Set edge to specified height ***  
        SET MATRIX HEIGHT 1,x,z,height#  
    NEXT z  
NEXT x  
REM *** Update screen ***  
UPDATE MATRIX 1  
REM *** Move camera using cursor keys ***  
DO  
    CONTROL CAMERA USING ARROWKEYS 0, 1, 1  
LOOP  
REM *** End program ***  
WAIT KEY  
END  
  
FUNCTION SetUpScreen()  
    SET DISPLAY MODE 1280,1024,32  
    REM *** Position camera ***  
    AUTOCAM OFF  
    POSITION CAMERA 0,50,100,-300  
ENDFUNCTION
```

Activity 45.5

Type in the above project (*matrix02.dbpro*) and test it.

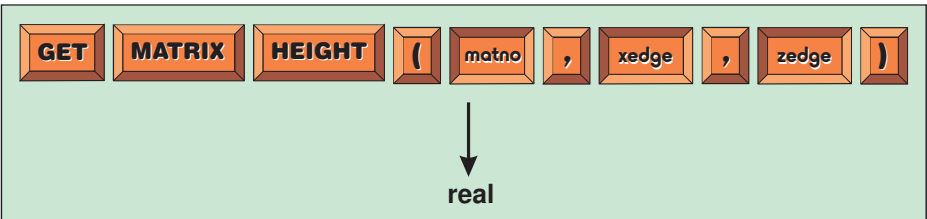
Modify the code so that the ramp effect goes from front to back rather than side-to-side.

Change the height increase to 50% on each increment of the X loop.

The GET MATRIX HEIGHT Statement

If you want to discover the exact height of any tile edge, you can do this using the GET MATRIX HEIGHT statement which has the format shown in FIG-45.12.

FIG-45.12
The GET MATRIX HEIGHT Statement



In the diagram:

matno is an integer value giving the matrix ID.

xedge is an integer value specifying a point along the x-axis.

zedge is an integer value specifying an edge along the z-axis.

The statement returns a real number representing how high above the base of the matrix the specified point is. For example, if we have previously executed the lines

```
MAKE MATRIX 1,150,100,3,2
SET MATRIX HEIGHT 1,3,2,10.5  `Height 10.5
UPDATE MATRIX 1
```

then the statement

```
PRINT GET MATRIX HEIGHT (1,3,12)
```

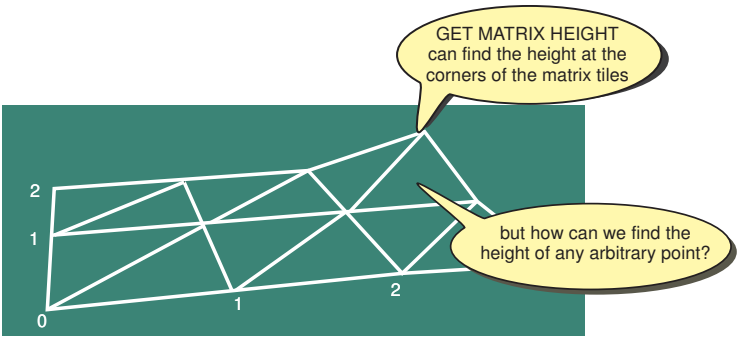
will display the value 10.5.

The GET GROUND HEIGHT Statement

The biggest limitation of the GET MATRIX HEIGHT statement is that it only allows the height at specific points to be returned. There’s no way of finding out the height half way along a tile (see FIG-45.13).

FIG-45.13

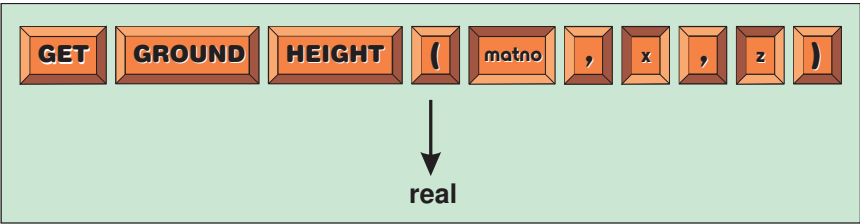
Finding the Height of any Point on the Matrix



The problem can be solved using the GET GROUND HEIGHT statement which returns the height at any point in the matrix. The statement has the format shown in FIG-45.14.

FIG-45.14

The GET GROUND HEIGHT Statement



In the diagram:

matno is an integer value giving the matrix ID.

x,y are real numbers which represent the position in the matrix whose height is to be returned. These values are given in world units - not matrix point numbers - and are relative to the bottom-left corner of the matrix.

If we created a matrix with the statements

```
MAKE MATRIX 1,150,100,3,2
SET MATRIX HEIGHT 1, 3, 2, 10.5   `Height 10.5
UPDATE MATRIX 1
```

then the statement

```
PRINT GET GROUND HEIGHT (1,150.0,100.0)
```

will return the value 10.5.

Remember, there are two main differences between GET MATRIX HEIGHT and GET GROUND HEIGHT:

- GET MATRIX HEIGHT can only give the height at a vertex;
GET GROUND HEIGHT can return the height of any position in the matrix.
- GET MATRIX HEIGHT identifies the position using edge numbers;
GET GROUND HEIGHT identifies the position using 3D world coordinates.

The program in LISTING-45.3 displays the height at various positions along the back edge of a matrix.

LISTING-45.3

Finding the Matrix Height
at any Point

```
SetUpScreen()
REM *** Create a grid (150 by 100) ***
MAKE MATRIX 1, 150, 100,3,2
REM *** Set the back corner to a height of 10.5 ***
SET MATRIX HEIGHT 1, 3,2, 10.5
REM *** Update matrix ***
UPDATE MATRIX 1
REM *** Get the height every 10 units ***
REM *** along the back edge of matrix ***
SYNC ON
FOR c = 0 TO 150 STEP 10
    SET CURSOR 0, 10
    PRINT "Height at (" ,c," ,",100," ) = ",GET GROUND HEIGHT(1,c,100)
    SYNC
    WAIT 500
NEXT c
REM *** End program ***
WAIT KEY
END

FUNCTION SetUpScreen()
    SET DISPLAY MODE 1280,1024,32
    REM *** Position camera ***
    AUTOCAM OFF
    POSITION CAMERA 0,200,300,-600
ENDFUNCTION
```

Activity 45.6

Type in and test the program given above (*matrix03.dbpro*).

Change the z-ordinate in the GET GROUND HEIGHT to 70.0 and check out the values displayed.

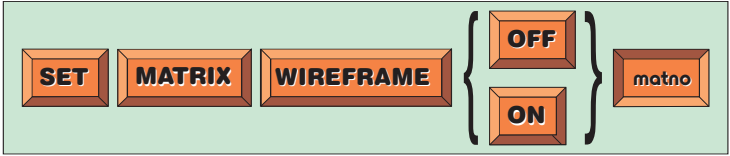
The SET MATRIX WIREFRAME Statement

Unlike other 3D objects, a grid appears in wireframe by default. We can toggle

between wireframe and solid mode using the SET MATRIX WIREFRAME statement which has the format shown in FIG-45.15.

FIG-45.15

The SET MATRIX WIREFRAME Statement



In the diagram:

- OFF, ON Use OFF to create a solid matrix; use ON to return to wireframe mode.
- matno* is an integer value giving the matrix ID.

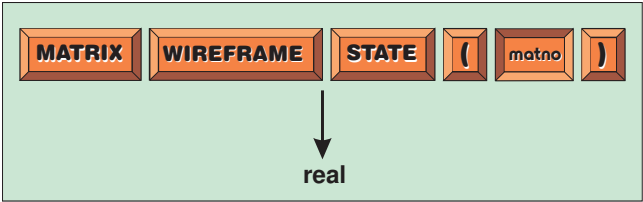
Activity 45.7
Modify your previous program so that the grid appears solid.

The MATRIX WIREFRAME STATE Statement

We can determine if a matrix is being displayed in wireframe or solid mode using the MATRIX WIREFRAME STATE statement. This statement has the format shown in FIG-45.16.

FIG-45.16

The MATRIX WIREFRAME STATE Statement



In the diagram:

- matno* is the integer value previously assigned to the matrix.
- This statement returns the value 1 if the matrix is being displayed in wireframe mode, otherwise zero is returned.

Adding Texture to the Matrix

The PREPARE MATRIX TEXTURE Statement

You will almost certainly want to add a texture to your grid. To do this you start by loading the required picture into an image object, just as we did for sprites and 3D primitives (spheres, cubes, etc.). However, the next step is a little different. The image to be used can be split up into rectangular areas known as **tiles** and only one of these tiles is used to texture the grid.

The PREPARE MATRIX TEXTURE statement is used to determine how the image is to be split up. This statement has the format shown in FIG-45.17.

FIG-45.17 The PREPARE MATRIX TEXTURE Statement



In the diagram:

- matno* is an integer value giving the matrix ID.
- imgno* is an integer value giving the ID of the image that is to be applied to the grid.
- xsects* is an integer value specifying the number of vertical sections in the image.
- zsects* is an integer value specifying the number of horizontal sections in the image.

For example, let's start by loading the image shown in FIG-45.18 below.

FIG-45.18

A Texture Image for the Matrix



If we begin by loading the image with the statement

```
LOAD IMAGE "B.bmp", 1
```

and then prepare the image, without dividing it into parts, with the line

```
PREPARE MATRIX TEXTURE 1, 1, 1, 1
```

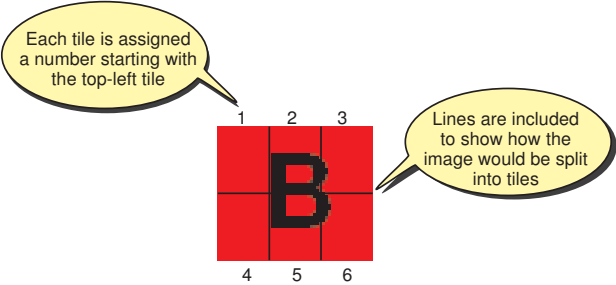
then the image will be taken as a single tile. However, if we use the statement

```
PREPARE MATRIX TEXTURE 1, 1, 3, 2
```

then the image will be split into tiles as shown in FIG-45.19.

FIG-45.19

Dividing an Image into Tiles



Activity 45.8

Reload *matrix02.dbpro* and add the line

```
LOAD IMAGE "B.bmp", 1
```

at an appropriate point in the program.

continued on next page

Activity 45.8 (continued)

Immediately before the line

```
UPDATE MATRIX 1
```

add the statement

```
PREPARE MATRIX TEXTURE 1,1,1,1
```

What effect does this have when you run your program?

Change the last line you added to

```
PREPARE MATRIX TEXTURE 1,1,3,2
```

How does this change the appearance of the grid?

As you can see from the results of the last Activity, the PREPARE MATRIX TEXTURE statement also applies one of the tiles in the image to every square in the grid. In the first instance, since the image had not been split into several tiles, the whole image is applied to each rectangle in the grid. In the second case, where the image had been split into six tiles, the top-left tile (tile 1) has been applied to every square, giving a completely red grid.

The FILL MATRIX Statement

If you'd rather fill the grid with a different tile from your image, you can use the FILL MATRIX statement. This statement also sets the height of each corner point in the whole grid. The statement has the format shown in FIG-45.20.

FIG-45.20

The FILL MATRIX
Statement



In the diagram:

matno

is an integer value giving the matrix ID.

height

is a real number specifying the y-ordinate of every point in the grid.

tileno

is an integer value specifying which tile from the prepared image is to be used to fill every rectangle in the grid.

This statement can only be used after a PREPARE MATRIX TEXTURE statement has been executed to set up the image being used by the grid.

For example, we could texture the complete grid with tile 2 from *B.bmp* and lift the whole grid to a height of 12.5 units using the lines:

```
PREPARE MATRIX TEXTURE 1,1,3,2  
FILL MATRIX 1,12.5,2
```

Since FILL MATRIX resets the height of the whole matrix, use this statement before using the SET MATRIX HEIGHT statement.

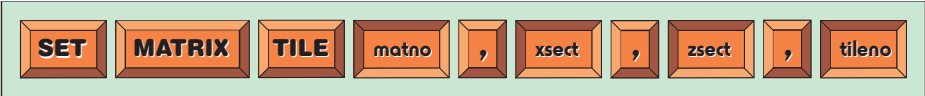
Activity 45.9

Modify your previous program to use tile 5 of the *B.bmp* image to texture your grid. Make sure the overall shape of the grid is unchanged.

The SET MATRIX TILE Statement

To get more control of texturing a grid we can use the SET MATRIX TILE statement to place a single tile from the image into one or more rectangles of the grid. This statement takes the format shown in FIG-45.21.

FIG-45.21 The SET MATRIX TILE Statement



In the diagram:

- matno* is an integer value giving the matrix ID.
- xsect* is an integer value specifying a section along the x-axis.
- zsect* is an integer value specifying a section along the z-axis.
- tileno* is an integer value representing the image tile to be used as the specified rectangle's texture.

For example, let's assume we've created a 5 by 3 grid using the statement

```
MAKE MATRIX 1,5,3
```

and split an image into 3 by 2 tiles with the lines

```
LOAD IMAGE "B.bmp", 1
PREPARE MATRIX TEXTURE 1,1,3,2
```

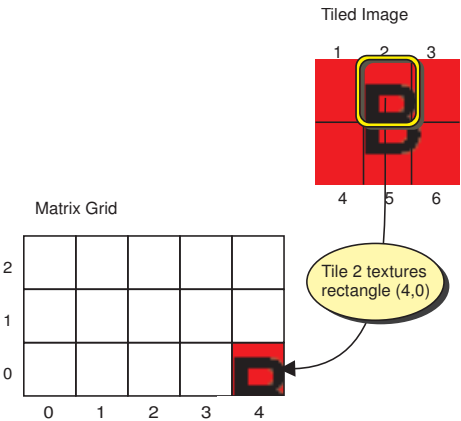
then we could place tile 2 of the image in rectangle (4,0) using the statement

```
SET MATRIX TILE 1,4,0,2
```

The result of this is shown in FIG-45.22.

FIG-45.22

The Effect of the SET
MATRIX TILE
Statement



The program in LISTING-45.4 demonstrates the effect of this statement.

LISTING-45.4

Texturing a Single
Rectangle

```
SetUpScreen()  
REM *** Create 5 by 3 grid ***  
MAKE MATRIX 1,100, 100,5,3  
REM *** Load image and tile as 3 by 2 ***  
LOAD IMAGE "B.bmp",1  
PREPARE MATRIX TEXTURE 1,1,3,2  
REM *** Load tile 2 into rectangle (4,0) ***  
SET MATRIX TILE 1,4,0,2  
REM *** Update display ***  
UPDATE MATRIX 1  
DO  
  CONTROL CAMERA USING ARROWKEYS 0, 1, 1  
LOOP  
REM *** End program ***  
END  
  
FUNCTION SetUpScreen()  
  SET DISPLAY MODE 1280,1024,32  
  AUTOCAM OFF  
  POSITION CAMERA 0,200,300,-600  
ENDFUNCTION
```

Activity 45.10

Type in and test the program given above (*matrix04.dbpro*).

Notice that, as well as loading the specified tile to the required rectangle in the grid, the remainder of the grid is red. This is because tile 1 is always used as default to fill the grid after the PREPARE MATRIX TEXTURE statement is executed.

Activity 45.11

Modify your last program so that tile 5 of the same image is placed in rectangle (2,1).

Modify the program again so that all six image tiles are stored over six rectangles. The rectangles should be placed as follows:

Tile	Rectangle
1	(1,2)
2	(2,2)
3	(3,2)
4	(1,1)
5	(2,1)
6	(3,1)

To spread a single image over the whole grid without repetition of the image, all you need to do is divide the image up so that it has one tile for every rectangle in the grid.

So if we create the grid with the line

```
MAKE MATRIX 1, 200,200,20,20
```

then we need to create our image with the lines:

```
LOAD IMAGE "F.bmp",1  
PREPARE MATRIX TEXTURE 1,1,20,20
```

Now we need to keep a tile count

```
tileno = 1
```

and then copy each tile to the correct rectangle

```
FOR x = 19 TO 0 STEP -1
  FOR z = 0 TO 19
    SET TEXTURE TILE 1,x,z,tileno
```

and move on to the next tile:

```
INC tileno
```

If we put all this together, we get the effect shown in FIG-45.23.

FIG-45.23

Spreading a Single
Image Over the Whole
Matrix



The complete program is shown in LISTING-45.5.

LISTING-45.5

Spreading a Single Image
Over the Whole Matrix

```
SetUpScreen()
REM *** Create grid 20 by 30 rectangles ***
MAKE MATRIX 1,200,300,20,30
REM *** Load image and divide into 20 by 30 tiles ***
LOAD IMAGE "F.bmp",1,0
PREPARE MATRIX TEXTURE 1,1,20,30
REM *** Copy each tile onto the grid starting at back-left ***
tileno = 1
FOR z =29 TO 0 STEP -1
  FOR x = 0 TO 19
    SET MATRIX TILE 1, x, z, tileno
    INC tileno
  NEXT x
NEXT z
REM *** Update grid ***
UPDATE MATRIX 1
REM *** Use camera to move about ***
DO
  CONTROL CAMERA USING ARROWKEYS 0, 1, 1
LOOP
REM *** End program ***
END

FUNCTION SetUpScreen()
  SET DISPLAY MODE 1280,1024,32
  AUTOCAM OFF
  POSITION CAMERA 0,100,-300
ENDFUNCTION
```

Activity 45.12

Type in and test the program given above (*matrix05.dbpro*).
Notice that it is necessary to start at the “back” of the grid (with $z = 29$) if the image is to be correctly mapped onto the matrix.

Change the texture image used to *grass.jpg* and randomly modify the points in the grid to vary up to 30 units in height.

The SET TEXTURE TRIM Statement

Very occasionally, you may not want the whole of a tile to be placed in a grid rectangle. Using the SET TEXTURE TRIM statement you can shave a bit off the edges of the image tile being used and thereby reduce what part of the image is actually used to texture a rectangle in the grid. The statement has the format shown in FIG-45.24.

FIG-45.24

The SET TEXTURE TRIM Statement



In the diagram:

matno

is an integer value giving the matrix ID.

xtrim

is a real value specifying how much is to be trimmed along the x edges of the tile image.

ztrim

is a real number specifying how much is to be trimmed along the z edges of the tile image.

This is a rather strange command to get your head round, so the program in LISTING-45.6 demonstrates its use by incrementing the trim factor along the x edge every time you press a key.

LISTING-45.6

Trimming the Texture Image

```
SetUpScreen()  
REM *** Set up grid with texture ***  
MAKE MATRIX 1,40,40,2,2  
LOAD IMAGE "trimmer.bmp",1,1  
PREPARE MATRIX TEXTURE 1,1,2,2  
REM *** Position grid ***  
POSITION MATRIX 1, 0,0,0  
REM *** Set xtrim value to zero ***  
xoff# = 0.0  
DO  
    REM *** Set the trim on the grid ***  
    SET MATRIX TRIM 1,xoff#,0.0  
    REM *** Fill every rectangle with same trimmed tile ***  
    FILL MATRIX 1,0,2  
    UPDATE MATRIX 1  
    REM *** Wait for key press ***  
    WAIT KEY  
    REM *** Make the offset larger ***  
    xoff# = xoff# + 0.01  
LOOP  
REM *** End program ***  
END  
  
FUNCTION SetUpScreen()  
    SET DISPLAY MODE 1280,1024,32  
    AUTOCAM OFF  
    POSITION CAMERA 0,100,-300  
ENDFUNCTION
```

Activity 45.13

Type in and test the program given above (*matrix06.dbpro*), observing how the texture of the grid changes as the *xoff#* value is changed.

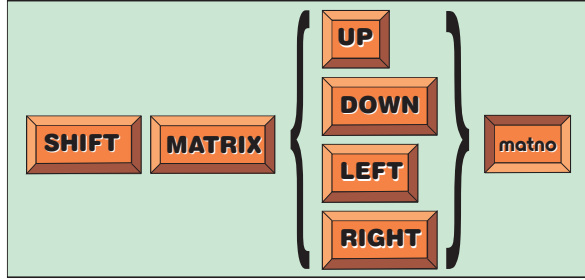
Modify the program so that both the *xtrim* and *ztrim* values are changed.

The SHIFT MATRIX Statement

Each rectangle in a grid can be shifted one position using the SHIFT MATRIX statement. This statement has several options as shown in FIG-45.25.

FIG-45.25

The SHIFT MATRIX
Statement



In the diagram:

UP, DOWN, LEFT, RIGHT

Choose the keyword corresponding to the direction in which the matrix is to be moved.

matno

is an integer value giving the matrix ID.

For example, the statement SHIFT MATRIX DOWN 1 has the effect of moving every rectangle in the matrix one position backwards within the matrix. Hence, rectangle (0,0) would move to position (0,1). The rectangles currently at the back of the grid are moved round to the front. All the information in that grid - the height of its corners and its texture are moved along with the rectangle. The overall effect is to give the impression that the surface is moving under the camera. The program in LISTING-45.7 demonstrates this effect.

LISTING-45.7

Shifting the Matrix
Texture

```
SetUpScreen()  
REM *** Set up grid with texture ***  
MAKE MATRIX 1,2000,3000,20,30  
LOAD IMAGE "grass.jpg",2  
PREPARE MATRIX TEXTURE 1,2,20,30  
tileno = 1  
FOR z =29 TO 0 STEP -1  
  FOR x = 0 TO 19  
    SET MATRIX TILE 1, x, z, tileno  
    INC tileno  
  NEXT x  
NEXT z  
RANDOMIZE MATRIX 1, 20  
REM *** Update grid on screen ***  
UPDATE MATRIX 1  
REM *** Shift matrix texturing ***  
DO  
  CONTROL CAMERA USING ARROWKEYS 0, 5, 1  
  REM *** Shift matrix one position ***  
  SHIFT MATRIX DOWN 1  
  UPDATE MATRIX 1  
  WAIT 50  
LOOP  
REM *** End program ***  
END  
  
FUNCTION SetUpScreen()  
  SET DISPLAY MODE 1280,1024,32  
  AUTOCAM OFF  
  POSITION CAMERA 0,100,-300  
ENDFUNCTION
```

Activity 45.14

Type in and test the program given in LISTING-45.7 (*matrix07.dbpro*).

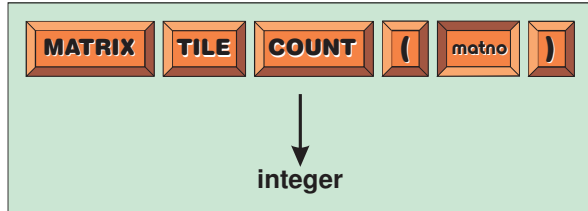
Modify the direction of shift to try out each of the other three options.

The MATRIX TILE COUNT Statement

The number of tiles available to a matrix - in other words, the total number of rectangles the prepared image has been split into by the PREPARE MATRIX TEXTURE statement, can be returned using the MATRIX TILE COUNT statement which has the format shown in FIG-45.26.

FIG-45.26

The MATRIX TILE
COUNT Statement



In the diagram:

matno is an integer value giving the matrix ID.

For example, if we had prepared an image for matrix 1 with the statement

```
PREPARE MATRIX TEXTURE 1, 2, 20, 30
```

then the line

```
PRINT MATRIX TILE COUNT(1)
```

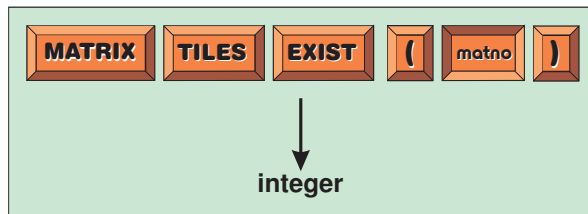
would display the value 600 (this being the result of 20 x 30).

The MATRIX TILES EXIST Statement

You can check that an image has been prepared for a specific matrix using the MATRIX TILES EXIST statement which has the format shown in FIG-45.27.

FIG-45.27

The MATRIX TILES
EXIST Statement



In the diagram:

matno is an integer value giving the matrix ID.

The statement returns 1 if a PREPARE MATRIX TEXTURE statement has already been executed for the matrix specified, otherwise zero is returned.

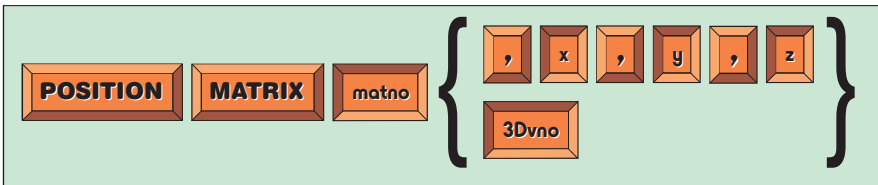
Positioning the Matrix in 3D Space

The POSITION MATRIX Statement

When a matrix is first created, its bottom-left corner is placed at position (0,0,0). Although the FILL MATRIX statement allows us to move the matrix in the y-axis, to position the whole matrix anywhere in 3D space, we need to use the POSITION MATRIX statement which has the format shown in FIG-45.28.

FIG-45.28

The POSITION MATRIX Statement



In the diagram:

matno is an integer value giving the matrix ID.

x,y,z are a sequence of real values specifying where the bottom-left corner of the grid is to be positioned.

3Dvno is an integer value previously assigned to a 3D vector object. The values in this object are used to position the matrix.

For example, the line

```
POSITION MATRIX 1,50,20,-40
```

would place the bottom-left corner of the grid at position (50,20,-40).

The program in LISTING-45.8 creates a matrix and a cube. The cube's centre is at position (0,0,0) and exists only to show the position of the 3D origin as the matrix is moved. The matrix is moved from its original position when the user presses a key.

LISTING-45.8

Positioning the Matrix

```
SetUpScreen()

REM *** Create grid 20 by 30 rectangles ***
MAKE MATRIX 1,200,300,20,30

REM *** Add random heights to matrix ***
RANDOMIZE MATRIX 1, 20

REM *** Create reference object ***
MAKE OBJECT CUBE 1, 10

REM *** Prepare matrix image ***
LOAD IMAGE "grass.jpg",1
PREPARE MATRIX TEXTURE 1,1,1,1

REM *** Texture matrix ***
SET MATRIX TEXTURE 1,1,1
```

continued on next page

LISTING-45.8

(continued)

Positioning the Matrix

```
REM *** Move matrix to new location ***  
WAIT KEY  
POSITION MATRIX 1,50,20,-40  
  
REM *** End program ***  
WAIT KEY  
END  
  
FUNCTION SetUpScreen()  
  SET DISPLAY MODE 1280,1024,32  
  AUTOCAM OFF  
  POSITION CAMERA 100,50,-300  
ENDFUNCTION
```

Activity 45.15

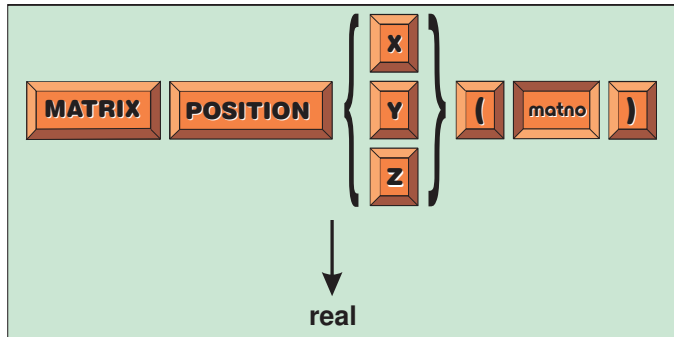
Type in and test the program in LISTING-45.8 (*matrix08.dbpro*).

The MATRIX POSITION Statement

Since a matrix object can be moved to any position within 3D space, DarkBASIC Pro offers a way to discover that position using the MATRIX POSITION statement. The format for the statement is shown in FIG-45.29.

FIG-45.29

The MATRIX POSITION
Statement



In the diagram:

X , Y , Z

Choose the option appropriate for the ordinate required.

matno

is an integer value giving the matrix ID.

The statement returns a real number representing the position of the bottom-left corner of the specified matrix. For example, if the bottom-left corner of matrix 1 is positioned at location (20,30,40) with the statement **POSITION MATRIX 1, 20, 30, 40** then the lines

```
PRINT "x-ord : ",MATRIX POSITION X(1)  
PRINT "y-ord : ",MATRIX POSITION Y(1)  
PRINT "z-ord : ",MATRIX POSITION Z(1)
```

will display the output:

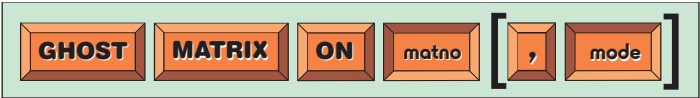
```
x-ord : 20  
y-ord : 30  
z-ord : 40
```


The GHOST MATRIX ON Statement

The transparency of a matrix can be modified (assuming your graphics card can handle this effect) using the GHOST MATRIX ON statement. This statement has the format shown in FIG-45.30.

FIG-45.30

The GHOST MATRIX
ON Statement



In the diagram:

matno is an integer value giving the matrix ID.

mode is an integer value between 0 and 5 which specifies the transparency mode to be used.

If the *mode* value is omitted, the matrix is displayed in standard transparency mode; when included, the *mode* parameter allows different transparency effects to be produced.

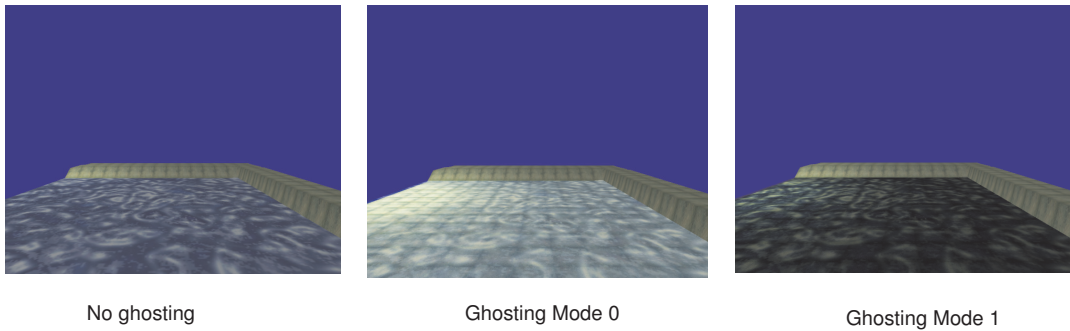
The program in LISTING-45.9 demonstrates the effects of the various mode settings by placing a matrix representing water just above a terrain object and switching modes each time a key is pressed. FIG-45.31 shows the display created by various modes.

LISTING-45.9

Creating a Transparent
Matrix

```
SetUpScreen()  
REM *** Change to point light ***  
SET POINT LIGHT 0,0,100,500  
REM *** Create first terrain object ***  
MAKE TERRAIN 1,"terraintest.bmp"  
LOAD IMAGE "grass2.jpg",1  
TEXTURE TERRAIN 1,1,  
POSITION TERRAIN 1,0,0,1023  
REM *** Set up matrix with water texture ***  
MAKE MATRIX 1,900,900,1,1  
LOAD IMAGE "030.bmp",2  
PREPARE MATRIX TEXTURE 1,2,1,1  
REM *** Position matrix above terrain ***  
POSITION MATRIX 1,0,5,0  
REM *** Use each transparency mode ***  
FOR c = 0 TO 5  
  GHOST MATRIX ON 1,c  
  REM *** Update grid on screen ***  
  UPDATE MATRIX 1  
  WAIT KEY  
NEXT c  
REM *** End program ***  
WAIT KEY  
END  
  
FUNCTION SetUpScreen()  
  SET DISPLAY MODE 1280,1024,32  
  AUTOCAM OFF  
  POSITION CAMERA 550,200,-300  
ENDFUNCTION
```

FIG-45.31 Varying Matrix Transparency



Activity 45.16

Type in and test the program given above (*matrix09.dbpro*).

The GHOST MATRIX OFF Statement

The transparency effect, if activated using GHOST MATRIX ON, can be switched off again using the GHOST MATRIX OFF statement which has the format shown in FIG-45.32.

FIG-45.32

The GHOST MATRIX
OFF Statement



In the diagram:

matrno

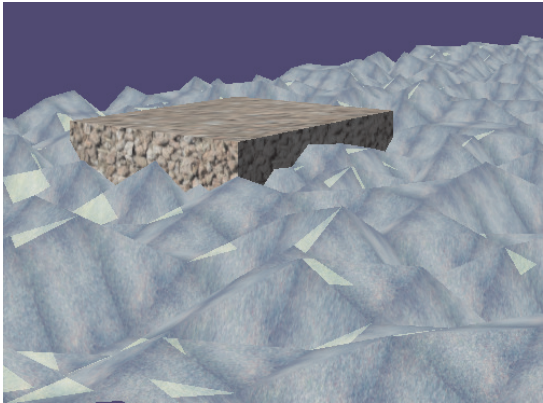
is an integer value giving the matrix ID.
The matrix should have previously been
made transparent using the GHOST MATRIX ON
statement.

The SET MATRIX PRIORITY Statement

If a program displays several 3D objects on screen, then these are drawn one after another to create the complete scene. Matrix objects are normally drawn first, but there are times when this gives the wrong effect. For example, in FIG-45.33 a cube is embedded in a transparent matrix, but you cannot see through the matrix to the bottom section of the cube.

FIG-45.33

Transparency Issues



The effect of embedding an object within a transparent matrix is shown in LISTING-45.10.

LISTING-45.10

Demonstrating Problems
with Transparency

```
SetUpScreen()  
REM *** Create a textured, transparent grid ***  
MAKE MATRIX 1, 2000, 2000,100,100  
LOAD IMAGE "grass1.jpg",1  
PREPARE MATRIX TEXTURE 1,1,1,1  
RANDOMIZE MATRIX 1, 40  
GHOST MATRIX ON 1  
UPDATE MATRIX 1  
REM *** Embed a cube in the matrix ***  
MAKE OBJECT CUBE 1,100  
LOAD IMAGE "wall_U5_01.jpg",2  
TEXTURE OBJECT 1,2  
POSITION OBJECT 1, 200,-10,200  
REM *** Allow user to move camera ***  
DO  
    CONTROL CAMERA USING ARROWKEYS 0,5,1  
LOOP  
REM *** End program ***  
END  
  
FUNCTION SetUpScreen()  
    SET DISPLAY MODE 1280,1024,32  
    AUTOCAM OFF  
    POSITION CAMERA 500,100,-100  
ENDFUNCTION
```

Activity 45.17

Type in the program above (*matrix10.dbpro*) and observe how the cube is displayed.

We can force a matrix object to be drawn last by using the SET MATRIX PRIORITY statement. This will create the desired transparency effect. The SET MATRIX PRIORITY has the format shown in FIG-45.34.

FIG-45.34

The SET MATRIX
PRIORITY Statement



In the diagram:

matno is an integer value giving the matrix ID.

prtyflg is set to zero if the matrix is to be drawn first and 1 (the default) if the matrix is to be drawn last.

Activity 45.18

In your last program, add the line

```
SET MATRIX PRIORITY 1,0
```

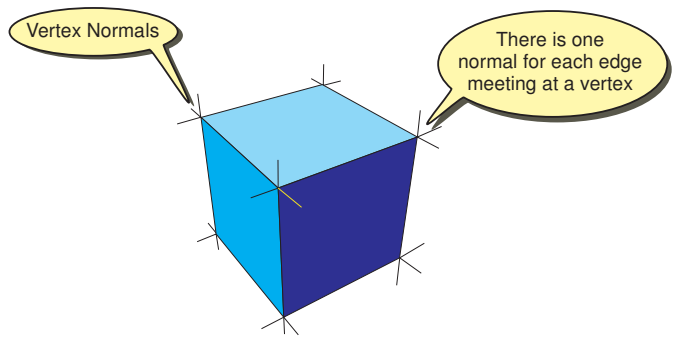
just before the MATRIX UPDATE statement and observe what difference this makes to the display.

The SET MATRIX NORMAL Statement

As we saw in earlier chapters, every vertex in a 3D shape has an invisible line known as a **vertex normal** which is used in lighting calculations (see FIG-45.35).

FIG-45.35

Vertex Normals



Since several triangles within a shape may share the same vertex, a single point may have many vertex normals. The angle and length of a normal has a direct effect on how a 3D object is lit. By changing a normal, we create an immediate change to how light is reflected at that point on the object to which the normal is attached.

The SET MATRIX NORMAL statement allows us to modify the length and direction of matrix normals. The statement requires the vertex and end point of the modified normal to be specified. The statement format is shown in FIG-45.36.

FIG-45.36 The SET MATRIX NORMAL Statement



In the diagram:

<i>matno</i>	is an integer value giving the matrix ID.
<i>xedge</i>	is an integer value specifying an edge along the x-axis.
<i>zedge</i>	is an integer value specifying an edge along the z-axis.
<i>xedge,zedge</i>	are a pair of integer values which together identify the location of the normal that is to be modified.
<i>nx,ny,nz</i>	are three integer values representing the new position of the unattached end of the vertex normal.

The program in LISTING-45.11 creates a single tile matrix and modifies a vertex normal each time a key is pressed.

LISTING-45.11

Modifying Matrix
Normals

```
SetUpScreen()  
REM *** Set up 200 by 100 grid (20 by 10 tiles) ***  
MAKE MATRIX 1,200, 100,1,1  
REM *** Change to point light ***  
SET POINT LIGHT 0,0,100,500  
REM *** Load image ***  
LOAD IMAGE "F.bmp",1  
REM *** Prepare image ***  
PREPARE MATRIX TEXTURE 1,1,1,1  
REM *** Modify the normal at each vertex ***  
FOR x = 0 TO 1  
  FOR z = 0 TO 1  
    SET MATRIX NORMAL 1, x, z, 0.0, 10.2, 0.0  
    UPDATE MATRIX 1  
    WAIT 1000  
  NEXT z  
NEXT x  
REM *** End program ***  
WAIT KEY  
END  
  
FUNCTION SetUpScreen()  
  SET DISPLAY MODE 1280,1024,32  
  AUTOCAM OFF  
  POSITION CAMERA 100,100,-200  
ENDFUNCTION
```

Activity 45.19

Type in the program above (*matrix11.dbpro*) and find out what effect the changes to the normal have on the way the matrix is lit.

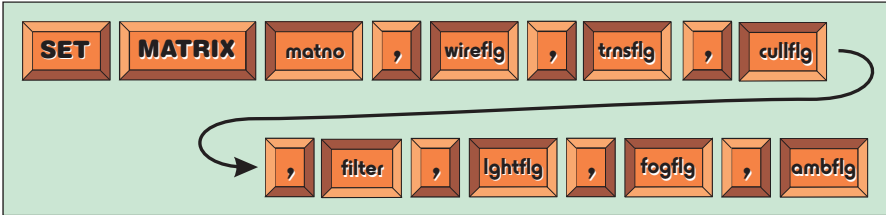
What effects are produced by varying the *nx*, *ny*, and *nz* values (use both positive and negative values)?

The SET MATRIX Statement

The SET MATRIX statement is by far the most powerful of the matrix commands, allowing you to set several matrix characteristics in a single line. For example, we can use it to switch wireframe mode on or off, set any black areas on the textured grid to become transparent, and change the way the grid’s surface reflects any light. The statement has the format shown in FIG-45.37.

FIG-45.37

The SET MATRIX
Statement



In the diagram:

- | | |
|----------------|--|
| <i>matno</i> | is an integer value giving the matrix ID. |
| <i>wireflg</i> | 0 - displays the grid in wireframe mode
1 - displays the grid in solid mode |
| <i>trnsflg</i> | 0 - grid is opaque.
1 - any area with black texturing becomes transparent. |

<i>cullflg</i>	0	-	hidden surfaces are drawn.
	1	-	hidden surfaces are not drawn.
<i>filter</i>	0	-	no texture mipmapping.
	1	-	no texture smoothing.
	2	-	uses linear filtering.
<i>lghtflg</i>	0	-	lights reflected normally.
	1	-	lights reflection reduced.
<i>fogflg</i>	0	-	fog does not affect grid.
	1	-	fog effects grid.
<i>ambflg</i>	0	-	ambient light is reflected from grid.
	1	-	ambient light reflection is reduced.

As you can see, there are many parameters to this statement. The best way to make sense of the effect it has on the matrix grid is to test each possible combination of parameter values. The program in LISTING-45.12 does exactly that, allowing the user to cycle through the values for each parameter.

LISTING-45.12

Testing the SET
MATRIX Statement

```

SetUpScreen()
REM *** Create a grid (1000 by 750) ***
MAKE MATRIX 1, 2000, 2000,100,100
REM *** Texture matrix ***
LOAD IMAGE "bwtexture.bmp",1
PREPARE MATRIX TEXTURE 1,1,1,1
REM *** Randomly adjust the y ordinates ***
REM *** to a maximum of 40 ***
RANDOMIZE MATRIX 1, 40
SET MATRIX PRIORITY 1,1
REM *** Update matrix ***
UPDATE MATRIX 1
REM *** Embed a cube in the matrix ***
MAKE OBJECT CUBE 1,100
LOAD IMAGE "bricks.jpg",2
TEXTURE OBJECT 1,2
POSITION OBJECT 1, 200,-10,200
REM *** Change to point light ***
SET POINT LIGHT 0,0,100,500
REM *** Create fog ***
FOG ON
FOG COLOR RGB(100,100,255)
FOG DISTANCE 1000
REM *** Set up parameter values ***
wire = 0
trans = 0
cull = 0
filter = 0
light = 0
fog = 0
amb = 0
REM *** Allow user to modify settings ***
SYNC ON
DO
    REM *** Display current settings ***
    SET CURSOR 0,10
    PRINT " 1 - Wireframe toggle (",wire,")"
    PRINT " 2 - Transparency toggle (",trans,")"
    PRINT " 3 - Cull Toggle (",cull,")"
    PRINT " 4 - Filter Increment (",filter,")"
    PRINT " 5 - Light Toggle (",light,")"

```

continued on next page

LISTING-45.12

(continued)

Testing the SET
MATRIX Statement

```
PRINT " 6 - Fog Toggle (",fog,")"
PRINT " 7 - Ambient Light Toggle (",amb,")"
REM *** Get parameter to be changed ***
choice = VAL(GetOption())
REM *** Change appropriate parameter's value ***
SELECT choice
  CASE 1
    wire = 1 - wire
  ENDCASE
  CASE 2
    trans = 1 - trans
  ENDCASE
  CASE 3
    cull = 1 - cull
  ENDCASE
  CASE 4
    filter = (filter + 1) mod 3
  ENDCASE
  CASE 5
    light = 1 - light
  ENDCASE
  CASE 6
    fog = 1 - fog
  ENDCASE
  CASE 7
    amb = 1 - amb
  ENDCASE
ENDSELECT
REM *** Show update matrix ***
SET MATRIX 1, wire,trans,cull,filter,light,fog, amb
UPDATE MATRIX 1
CONTROL CAMERA USING ARROWKEYS 0,1,1
SYNC
WAIT 100
LOOP
REM *** End program ***
END

FUNCTION SetUpScreen()
  SET DISPLAY MODE 1280,1024,32
  AUTOCAM OFF
  POSITION CAMERA 500,100,-100
ENDFUNCTION

FUNCTION GetOption()
  result$ = ""
  REPEAT
    result$ = INKEY$()
  UNTIL result$ <> ""
  WHILE INKEY$() <> ""
  ENDWHILE
ENDFUNCTION result$
```

Activity 45.20

Type in and test the program given in LISTING-45.12 (*matrix12.dbpro*).

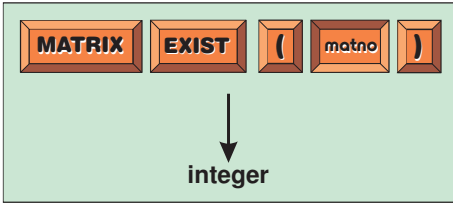
How do the *ambflg* and *lghtflg* values interact?

The MATRIX EXIST Statement

Since matrices can be destroyed as well as created, you can check to make sure that a specific matrix exists using the MATRIX EXIST statement as shown in FIG-45.38.

FIG-45.38

The MATRIX EXIST
Statement



In the diagram:

matno is an integer value giving the matrix ID.

The statement returns 1 if the specified matrix does exist, otherwise zero is returned. Hence, we might report the non-existence of matrix 1 with the statement:

```
IF MATRIX EXIST(1) <> 1
    PRINT "Matrix does not exist"
ENDIF
```

Summary

- A matrix object can be used to create a landscape.
- Use MAKE MATRIX to create a matrix object.
- A matrix is divided into a number of subdivisions along the x and z axes.
- Use RANDOMIZE MATRIX to randomly set the height of each vertex within a matrix.
- Use UPDATE MATRIX to update a matrix on-screen after changes have been made.
- Use SET MATRIX HEIGHT to modify the height of individual vertices within a matrix.
- Use GET MATRIX HEIGHT to discover the height of any vertex on a matrix.
- Use GET GROUND HEIGHT to discover the height on any arbitrary point on a matrix.
- A matrix is initially shown in wireframe mode.
- Use SET MATRIX WIREFRAME to toggle between wireframe and solid display modes.
- Use MATRIX WIREFRAME STATE to discover the current display mode of a matrix.
- Use PREPARE MATRIX TEXTURE to split an image into 1 or more sections. One of these sections will be used automatically to texture the matrix.
- Use FILL MATRIX to texture the matrix with a specified section of the image specified in PREPARE MATRIX TEXTURE.
- Use SET MATRIX TILE to place a section of a prepared image anywhere on a matrix.

- Use SET TEXTURE TRIM to fine tune the mapping of a texture to a matrix.
- Use SHIFT MATRIX to offset the rectangles that make up a grid.
- Use MATRIX TILE COUNT to discover how many sections a prepared image has been split into.
- Use MATRIX TILES EXIST to check that an image has been prepared for a matrix.
- Use POSITION MATRIX to reposition a matrix object. The bottom-left corner of the matrix is placed at the position specified.
- Use MATRIX POSITION to discover the current position of a matrix object.
- Use GHOST MATRIX ON to create a transparency effect on a matrix.
- Use GHOST MATRIX OFF to switch off transparency.
- Use SET MATRIX PRIORITY to modify the order in which matrices are drawn on the screen.
- Use SET MATRIX NORMAL to modify the lighting of a matrix at a specific point.
- Use SET MATRIX to set several matrix properties in a single command.
- Use MATRIX EXIST statement to determine if a matrix of a specified ID exists.

Solutions

Activity 45.1

```
MAKE MATRIX 2,1000,500,10,5
```

Activity 45.2

```
REM *** Set screen resolution ***  
SET DISPLAY MODE 800, 600, 16
```

```
REM *** Position camera ***  
AUTOCAM OFF  
POSITION CAMERA 0, 500,300,-600
```

```
REM *** Create a grid (1000 by 500) ***  
MAKE MATRIX 1, 1000,500,100,50
```

```
REM *** Randomly adjust height ***  
REM *** to a maximum of 20 ***  
RANDOMIZE MATRIX 1, 20
```

```
REM *** Update matrix ***  
UPDATE MATRIX 1
```

```
REM *** End program ***  
WAIT KEY  
END
```

Activity 45.3

```
REM *** Set screen resolution ***  
SET DISPLAY MODE 800, 600, 16  
REM *** Position camera ***  
AUTOCAM OFF  
POSITION CAMERA 0, 500,300,-600  
REM *** Create a grid (1000 by 500) ***  
MAKE MATRIX 1, 1000,500,100,50  
REM *** Randomly adjust height ***  
REM *** to a maximum of 20 ***  
RANDOMIZE MATRIX 1, 20  
REM *** Update matrix ***  
UPDATE MATRIX 1  
REM *** Allow user to move camera ***  
DO  
    CONTROL CAMERA USING ARROWKEYS 0,5,1  
LOOP  
REM *** End program ***  
WAIT KEY  
END
```

```
FUNCTION SetUpScreen()  
    SET DISPLAY MODE 1280,1024,32  
    REM *** Position camera ***  
    AUTOCAM OFF  
    POSITION CAMERA 0,0,100,-300  
ENDFUNCTION
```

Activity 45.4

```
REM *** Set screen resolution ***  
SET DISPLAY MODE 800, 600, 16  
  
REM *** Position camera for a good view ***  
AUTOCAM OFF  
POSITION CAMERA 0, 500,300,-600
```

```
REM *** Create a grid (1000 by 500) ***  
MAKE MATRIX 1, 1000,500,100,50
```

```
REM ** Lift (8,3) to 25.7 ***  
SET MATRIX HEIGHT 1, 8, 3, 25.7  
REM *** Sink (0,1) to -12.5 ***
```

```
SET MATRIX HEIGHT 1, 0, 1, -12.5
```

```
REM *** Update matrix ***  
UPDATE MATRIX 1
```

```
REM *** Allow user to move camera ***  
DO  
    CONTROL CAMERA USING ARROWKEYS 0,5,1  
LOOP
```

```
REM *** End program ***  
WAIT KEY  
END
```

Activity 45.5

```
REM *** Set screen resolution ***  
SET DISPLAY MODE 800,600,16
```

```
REM *** Set up 200 by 100 grid ***  
MAKE MATRIX 1,200, 100,20,10
```

```
REM *** Set initial height ***  
height#=0.5
```

```
REM *** FOR each Z edge DO ***  
FOR z = 0 TO 10  
    REM *** Increase height by 50% ***  
    height# = height# *1.5  
    REM *** FOR each X edge DO ***  
    FOR x = 0 TO 20  
        REM *** Set height of edge ***  
        SET MATRIX HEIGHT 1, X, Z, height#  
    NEXT x  
NEXT z  
REM *** Update screen ***  
UPDATE MATRIX 1
```

```
REM *** Position camera ***  
AUTOCAM OFF  
POSITION CAMERA 0, 0,100,-300
```

```
REM *** Allow user to move camera ***  
DO  
    CONTROL CAMERA USING ARROWKEYS 0, 1, 1  
LOOP
```

```
REM *** End program ***  
WAIT KEY  
END
```

Activity 45.6

```
REM *** Set screen resolution ***  
SET DISPLAY MODE 1280,1024,16
```

```
REM *** Position camera ***  
AUTOCAM OFF  
POSITION CAMERA 0, 100,100,-200
```

```
REM *** Create a grid (150 by 100)***  
MAKE MATRIX 1, 150,100,3,2
```

```
REM *** Set back corner height to 10.5 ***  
SET MATRIX HEIGHT 1, 3,2, 10.5
```

```
REM *** Update matrix ***  
UPDATE MATRIX 1
```

```
REM *** Get the height every 10 units ***  
REM *** along 70.0 line of matrix ***  
SYNC ON
```

```

FOR c = 0 TO 150 STEP 10
  SET CURSOR 0, 10
  PRINT "Height at (" ,c," ",70," ) = ",
    GET GROUND HEIGHT(1,c,70)
  SYNC
  WAIT 500
NEXT c

REM *** End program ***
WAIT KEY
END

```

Activity 45.7

```

REM *** Set screen resolution ***
SET DISPLAY MODE 1280,1024,16

REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 0, 100,100,-200

REM *** Create a grid (150 by 100)***
MAKE MATRIX 1, 150,100,3,2

REM *** Set back corner height to 10.5 ***
SET MATRIX HEIGHT 1, 3,2, 10.5

REM *** Make matrix solid ***
SET MATRIX WIREFRAME OFF 1

REM *** Update matrix ***
UPDATE MATRIX 1

REM *** Get the height every 10 units ***
REM *** along 70.0 line of matrix ***
SYNC ON
FOR c = 0 TO 150 STEP 10
  SET CURSOR 0, 10
  PRINT "Height at (" ,c," ",70," ) = ",
    GET GROUND HEIGHT(1,c,70)
  SYNC
  WAIT 500
NEXT c

REM *** End program ***
WAIT KEY
END

```

Activity 45.8

```

REM *** Set screen resolution ***
SET DISPLAY MODE 800,600,16

REM *** Load texture image ***
LOAD IMAGE "B.bmp",1

REM *** Set up 200 by 100 grid ***
MAKE MATRIX 1,200, 100,20,10

REM *** Set initial height ***
height#=0.5

REM *** FOR each X edge DO ***
FOR z = 0 TO 1
  REM *** Increase height by 50% ***
  height# = height# *1.5
  REM *** FOR each Z edge DO ***
  FOR x = 0 TO 20
    REM *** Set height of edge ***
    SET MATRIX HEIGHT 1, X, Z, height#
  NEXT x
NEXT z

REM *** Texture matrix ***
PREPARE MATRIX TEXTURE 1,1,3,2

```

```

REM *** Update screen ***
UPDATE MATRIX 1

REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 0, 0,100,-300

REM *** Move camera using cursor keys ***
DO
  CONTROL CAMERA USING ARROWKEYS 0, 1, 1
LOOP

REM *** End program ***
WAIT KEY
END

```

The texture image has been split into 6 tiles (3 by 2). It is the first of these tiles (the top-left one) which is used to texture the whole grid.

Activity 45.9

```

REM *** Set screen resolution ***
SET DISPLAY MODE 800,600,16

REM *** Load texture image ***
LOAD IMAGE "B.bmp",1

REM *** Set up 200 by 100 grid ***
MAKE MATRIX 1,200, 100,20,10

REM *** Texture matrix ***
PREPARE MATRIX TEXTURE 1,1,3,2
FILL MATRIX 1, 0, 5

REM *** Set initial height ***
height#=0.5

REM *** FOR each X edge DO ***
FOR z = 0 TO 10
  REM *** Increase height by 50% ***
  height# = height# *1.5
  REM *** FOR each Z edge DO ***
  FOR x = 0 TO 20
    REM *** Set height of edge ***
    SET MATRIX HEIGHT 1, X, Z, height#
  NEXT x
NEXT z

REM *** Update screen ***
UPDATE MATRIX 1

REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 0, 0,100,-300

REM *** Move camera using cursor keys ***
DO
  CONTROL CAMERA USING ARROWKEYS 0, 1, 1
LOOP

REM *** End program ***
WAIT KEY
END

```

Notice that the PREPARE MATRIX TEXTURE statement has been moved so that the FILL MATRIX can be executed before SET MATRIX HEIGHT.

Activity 45.10

No solution required.

Activity 45.11

First modification:

```
REM *** Set screen resolution ***
SET DISPLAY MODE 1280, 1024, 16

REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 0, 200,300,-600

REM *** Create 5 by 3 grid ***
MAKE MATRIX 1,100, 100,5,3

REM *** Load image and tile as 3 by 2 ***
LOAD IMAGE "B.bmp",1
PREPARE MATRIX TEXTURE 1,1,3,2

REM *** Load tile 2 into rectangle (4,0)
***
SET MATRIX TILE 1,4,0,2

REM *** Load tile 5 into rectangle (2,1)
***
SET MATRIX TILE 1,2,1,5

REM *** Update display ***
UPDATE MATRIX 1

REM *** Get user control of camera ***
DO
    CONTROL CAMERA USING ARROWKEYS 0, 1, 1
LOOP

REM *** End program ***
WAIT KEY
END
```

Second modification:

```
REM *** Set screen resolution ***
SET DISPLAY MODE 1280, 1024, 16

REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 0, 200,300,-600

REM *** Create 5 by 3 grid ***
MAKE MATRIX 1,100, 100,5,3

REM *** Load image and tile as 3 by 2 ***
LOAD IMAGE "B.bmp",1
PREPARE MATRIX TEXTURE 1,1,3,2
REM *** Tile 1 - (1,2)
SET MATRIX TILE 1,1,2,1
REM *** Load tile 2 into rectangle (4,0)
***
SET MATRIX TILE 1,4,0,2
REM *** Tile 3 - (3,2) ***
SET MATRIX TILE 1, 3,2,3
REM *** Tile 4 - (1,1)
SET MATRIX TILE 1,1,1,4
REM *** Load tile 5 into rectangle (2,1)
***
SET MATRIX TILE 1,2,1,5
REM *** Tile 6 - (3,1)
SET MATRIX TILE 1,3,1,6
REM *** Update display ***
UPDATE MATRIX 1

REM *** Get user control of camera ***
DO
    CONTROL CAMERA USING ARROWKEYS 0, 1, 1
LOOP
REM *** End program ***
WAIT KEY
END
```

Activity 45.12

```
REM *** Set screen resolution ***
SET DISPLAY MODE 800,600,16

REM *** Position camera ***
AUTOCAM OFF
POSITION CAMERA 0, 0,100,-300

REM *** Create grid 20 by 30 rectangles
***
MAKE MATRIX 1,200,300,20,30

REM *** Set random height in matrix ***
RANDOMIZE MATRIX 1, 30

REM *** Divide image into 20 x 30 tiles
***
LOAD IMAGE "grass.jpg",1,0
PREPARE MATRIX TEXTURE 1,1,20,30

REM *** Copy each tile onto the grid ***
REM *** starting at back-left ***
tileno = 1
FOR z =29 TO 0 STEP -1
    FOR x = 0 TO 19
        SET MATRIX TILE 1, x, z, tileno
        INC tileno
    NEXT x
NEXT z

REM *** Update grid ***
UPDATE MATRIX 1

REM *** Use camera to move about ***
DO
    CONTROL CAMERA USING ARROWKEYS 0, 1, 1
LOOP

REM *** End program ***
WAIT KEY
END
```

Activity 45.13

```
REM *** Set screen resolution ***
SET DISPLAY MODE 800,600,16

REM *** and camera ***
AUTOCAM OFF
POSITION CAMERA 0, 0,100,-300

REM *** Set up grid with texture ***
MAKE MATRIX 1,40,40,2,2
LOAD IMAGE "trimmer.bmp",1,1
PREPARE MATRIX TEXTURE 1,1,2,2

REM *** Position grid ***
POSITION MATRIX 1, 0,0,0
REM *** Set xtrim and ztrim to zero ***
xoff# = 0.0
zoff# = 0.0
DO
    REM *** Set the trim on the grid ***
    SET MATRIX TRIM 1,xoff#,0.0
    REM *** Fill rectangles with tile ***
    FILL MATRIX 1,0,2
    UPDATE MATRIX 1
    REM *** Wait for key press ***
    WAIT KEY
    REM *** Make the offset larger ***
    xoff# = xoff# + 0.01
    zoff# = zoff# + 0.01
LOOP
REM *** End program ***
WAIT KEY
END
```

Activity 45.14

```
REM *** Set up grid with texture ***
MAKE MATRIX 1,2000,3000,20,30
LOAD IMAGE "grass.jpg",2
PREPARE MATRIX TEXTURE 1,2,20,30
tileno = 1
FOR z =29 TO 0 STEP -1
  FOR x = 0 TO 19
    SET MATRIX TILE 1, x, z, tileno
    INC tileno
  NEXT x
NEXT z
RANDOMIZE MATRIX 1, 20

REM *** Update grid on screen ***
UPDATE MATRIX 1

REM *** Use camera ***
AUTOCAM OFF
POSITION CAMERA 0, 0,100,-300
DO
  CONTROL CAMERA USING ARROWKEYS 0, 5, 1
  REM *** Shift matrix one position ***
  SHIFT MATRIX UP 1
  UPDATE MATRIX 1
  WAIT 50
LOOP

REM *** End program ***
END
```

Change

```
SHIFT MATRIX UP 1
```

to

```
SHIFT MATRIX LEFT 1
```

and later to

```
SHIFT MATRIX RIGHT 1
```

to test other directions.

Activity 45.20

lghtflg must be 1 before *ambflg* is effective.

Activity 45.15

No solution required.

Activity 45.16

No solution required.

Activity 45.17

The lower part of the cube can be seen through the transparent matrix.

Activity 45.18

The lower part of the cube can no longer be seen.

Activity 45.19

The lighting effect changes subtly for different values. Using negative values gives a much duller finish.

Manipulating Vertices

Accessing and Modifying a Vertex's Normal

Accessing and Modifying a Vertex's UV Settings

Capturing an Object's Vertex Data

Creating New Shapes by Modifying Vertex Coordinates

Modifying a Vertex's Coordinates

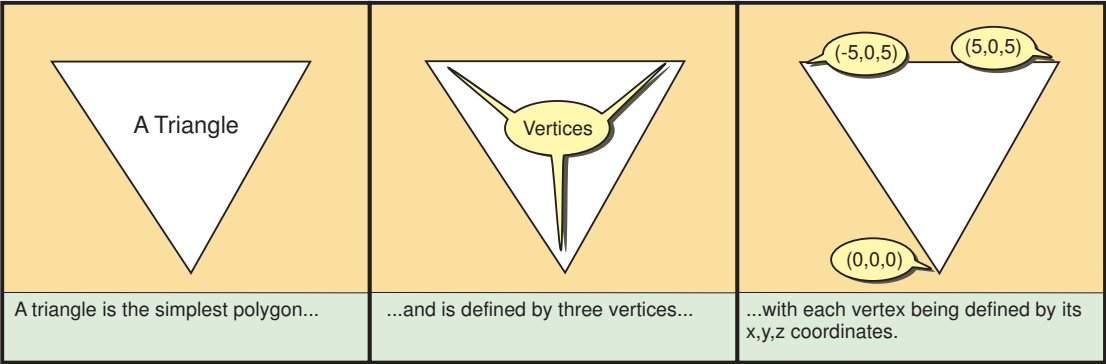
Reading a Vertex's Coordinates

Manipulating Vertices

Introduction

Every 3D object is constructed from a set of triangular polygons and each of the three vertices that make up a triangle are represented by a set of (x,y,z) coordinates (see FIG-46.1).

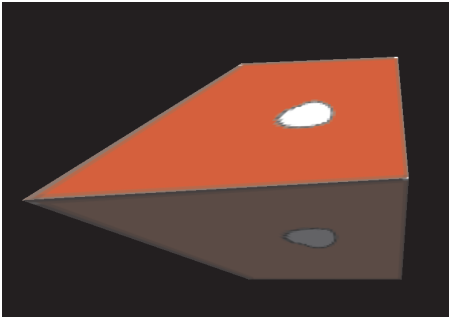
FIG-46.1 Defining a Triangular Polygon



The individual vetices of a 3D mesh or limb can be manipulated using a set of vertex data statements that are included in DarkBASIC Pro. For example, we could start with a cube and move one of its vertices to create a new shape (see FIG-46.2).

FIG-46.2

A Distorted Cube



The Statements

The LOCK VERTEXDATA FOR MESH Statement

Before we can begin to modify the vertex data of a mesh, we need to take a copy of the coordinates of the mesh's vertices. To capture the vertex data we use the LOCK VERTEXDATA FOR MESH statement which has the format shown in FIG-46.3.

FIG-46.3

The LOCK
VERTEXDATA FOR
MESH Statement



In the diagram:

meshno

is an integer value specifying the ID of the mesh whose vertex details are to be copied.

For example, we could create a simple triangle object with the statement

```
MAKE OBJECT TRIANGLE 1,0,0,0,-5,0,5,5,0,5
```

and then convert it to a mesh with the line:

```
MAKE MESH FROM OBJECT 1,1
```

From there we can save the new mesh's vertex data with the line

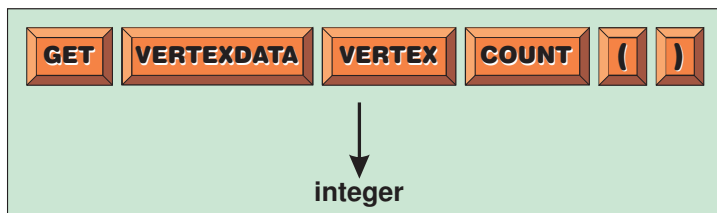
```
LOCK VERTEXDATA FOR MESH 1
```

The GET VERTEXDATA VERTEX COUNT Statement

A complex mesh may contain hundreds of vertices. The number of vertices in a set of locked vertex data can be discovered using the GET VERTEXDATA VERTEX COUNT which has the format shown in FIG-46.4.

FIG-46.4

The GET VERTEXDATA
INDEX COUNT Statement



The statement returns the number of vertices in the locked data.

For example, we could determine the number of vertices in our locked triangle (as given in the example above) using the line:

```
count = GET VERTEXDATA VERTEX COUNT()
```

The program in LISTING-46.1 brings together the example statements used earlier to create a triangle, convert it to a mesh, store the triangle's data and display the vertex count.

LISTING-46.1

Finding the Number of
Vertices in a Mesh

```
SetUpScreen()  
REM *** Create triangle ***  
MAKE OBJECT TRIANGLE 1,0,0,0,-5,0,5,5,0,5  
REM *** Convert it to a mesh ***  
MAKE MESH FROM OBJECT 1,1  
DELETE OBJECT 1  
REM *** Capture mesh vertex data ***  
LOCK VERTEXDATA FOR MESH 1  
REM *** Find out number of vertices in mesh ***  
count = GET VERTEXDATA VERTEX COUNT()  
REM *** Display number of vertices ***  
DO  
    SET CURSOR 100,100  
    PRINT count  
LOOP  
REM *** End program ***  
END  
  
FUNCTION SetUpScreen()  
    SET DISPLAY MODE 1280,1024,32  
    COLOR BACKDROP 0  
    BACKDROP ON  
    AUTOCAM OFF  
ENDFUNCTION
```

Activity 46.1

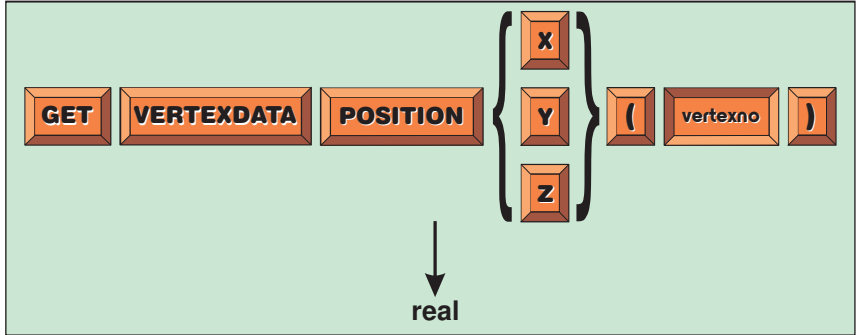
Type in and test the program given in LISTING-46.1 (*vertex01.dbpro*).

The GET VERTEXDATA POSITION Statement

We can find the coordinates of each locked vertex using the GET VERTEXDATA POSITION statement which has the format shown in FIG-46.5.

FIG-46.5

The GET
VERTEXDATA
POSITION Statement



In the diagram:

X,Y,Z

Use one of these options appropriate for the ordinate required.

vertexno

is an integer value giving the number of the vertex whose coordinates are to be retrieved. Note that the first vertex number is zero.

The triangle mesh we created in the first program will have vertices numbered, 0, 1, and 2. To retrieve the coordinates of vertex 0 we would use the lines:

```
x# = GET VERTEXDATA POSITION X(0)
y# = GET VERTEXDATA POSITION Y(0)
z# = GET VERTEXDATA POSITION Z(0)
```

Activity 46.2

Add the following lines at appropriate positions within your last program and hence display the coordinates of the second vertex (vertex 1) :

```
REM *** Get coordinates of second vertex ***
x# = GET VERTEXDATA POSITION X(1)
y# = GET VERTEXDATA POSITION Y(1)
z# = GET VERTEXDATA POSITION Z(1)

SET CURSOR 100,120
PRINT STR$(x#,0) , "  " , STR$(y#,0) , "  " , STR$(z#,0)
```

The next program (see LISTING-46.2) stores the coordinates of a triangles vertices in a set of arrays - one array for all the x-ordinates, one for the y-ordinates, and one for the z-ordinates. The coordinates of the vertices are then displayed. The program includes a routine (*Format\$()*) to create an aligned layout.

LISTING-46.2

Displaying the
Coordinates of an
Object's Vertices

```
SetUpScreen()

REM *** Create root object for model ***
MAKE OBJECT SPHERE 99,0

REM *** Create triangle ***
MAKE OBJECT TRIANGLE 1,0,0,0,-5,0,5,5,0,5

REM *** Convert it to a mesh ***
MAKE MESH FROM OBJECT 1,1
DELETE OBJECT 1

REM *** ADD mesh to model ***
ADD LIMB 99,1,1

REM *** Copy the vertex data ***
LOCK VERTEXDATA FOR MESH 1

REM *** Retrieve number of vertices in mesh ***
count = GET VERTEXDATA INDEX COUNT()

REM *** Create arrays to hold the coordinates of each vertex ***
DIM x#(count)
DIM y#(count)
DIM z#(count)

REM *** Store coords of each vertex in arrays ***
FOR c = 0 TO count-1
  x#(c) = GET VERTEXDATA POSITION X(c)
  y#(c) = GET VERTEXDATA POSITION Y(c)
  z#(c) = GET VERTEXDATA POSITION Z(c)
NEXT c

REM *** Display vertex count and coords ***
DO
  SET CURSOR 100,100
  PRINT "Number of vertices in mesh : ",count
  PRINT
  FOR c = 0 TO count-1
    SET CURSOR 100,120+c*20
    PRINT "Vertex number ",c,"          ",Format$(x#(c),2,0)," ",
      Format$(y#(c),2,0)," ",Format$(z#(c),2,0)
  NEXT c
LOOP

REM *** End program ***
END

FUNCTION SetUpScreen()
  SET DISPLAY MODE 1280,1024,32
  COLOR BACKDROP 0
  BACKDROP ON
  AUTOCAM OFF
  POSITION CAMERA 0,0,-20
  POINT CAMERA 0,0,0
ENDFUNCTION

FUNCTION Format$(v#,intpart,decpl)
  result$ = STR$(v#,decpl)
  result$ = SPACE$(intpart+decpl-LEN(result$))+result$
ENDFUNCTION result$
```

Activity 46.3

Type in and test the program given in LISTING-46.2 (*vertex02.dbpro*).

The SET VERTEXDATA POSITION Statement

We can alter the coordinates of a specified vertex using the SET VERTEXDATA POSITION statement which has the format shown in FIG-46.6.

FIG-46.6

The SET VERTEXDATA POSITION Statement



In the diagram:

vertexno

is an integer value giving the vertex number of the vertex whose coordinates are to be changed.

x,y,z

are a set of real numbers giving the new coordinates to be assigned to the specified vertex.

For example, we could change the coordinates of vertex 0 in our triangle to (5,5,5) using the line:

```
SET VERTEXDATA POSITION 0,5,5,5
```

However, this won't have any effect on the triangle on the screen. To do that we must unlock the vertex values.

The UNLOCK VERTEXDATA Statement

The UNLOCK VERTEXDATA statement allows the updated vertex data to be applied to the mesh or limb from which they were originally extracted. This statement has the format shown in FIG-46.7.

FIG-46.7

The UNLOCK
VERTEXDATA
Statement



The program in LISTING-46.3 adds a mesh as a limb to a model, modifies the mesh's vertex data, and then adds the modified mesh as a second limb; finally, the model is rotated.

LISTING-46.3

Modifying a Mesh's
Vertices

```
SetUpScreen ()

REM *** Create model root ***
MAKE OBJECT SPHERE 99,0

REM *** Make triangle mesh ***
MAKE OBJECT TRIANGLE 1,-1,0,0,-5,0,5,5,0,5
MAKE MESH FROM OBJECT 1,1
DELETE OBJECT 1

REM *** Add triangle as first limb ***
ADD LIMB 99,1,1

REM *** Modify mesh data ***
LOCK VERTEXDATA FOR MESH 1
SET VERTEXDATA POSITION 1,5,8,5
UNLOCK VERTEXDATA
```

continued on next page

LISTING-46.3
(continued)

Modifying a Mesh's
Vertices

```
REM *** Add modified mesh as second limb ***
ADD LIMB 99,2,1

REM *** Rotate model ***
DO
    TURN OBJECT LEFT 99,0.1
    WAIT 1
LOOP

REM *** End program ****
END

FUNCTION SetUpScreen()
    SET DISPLAY MODE 1280,1024,32
    COLOR BACKDROP RGB(255,255,100)
    BACKDROP ON
    AUTOCAM OFF
    POSITION CAMERA 0,0,-20
    POINT CAMERA 0,0,0
ENDFUNCTION
```

Activity 46.4

Type in and test the program in LISTING-46.3 (*vertex03.dbpro*).

The LOCK VERTEXDATA FOR LIMB Statement

Rather than retrieve vertex data relating to a mesh, it is also possible to capture the vertex data of a limb (which, after all, is simply a mesh which has been added to a model). This is done using the LOCK VERTEXDATA FOR LIMB statement which has the format shown in FIG-46.8.

FIG-46.8
The LOCK
VERTEXDATA FOR
LIMB Statement



In the diagram:

- objno* is an integer value specifying the ID of the root object of a model.
- limbno* is an integer value specifying the ID of the limb whose vertex data is to be copied.

Once the vertex data has been copied, the usual vertex data commands such as GET VERTEXDATA INDEX COUNT, GET VERTEXDATA POSITION and SET VERTEXDATA POSITION can be applied. When the UNLOCK VERTEXDATA is executed the affected limb within the model will be modified.

In the next program (see LISTING-46.4) a triangle is added as limb 1 within a model and then modified directly.

LISTING-46.4

Modifying Limb
vertices Directly

```
SetUpScreen()

REM *** Make triangular mesh ***
MAKE OBJECT TRIANGLE 1,0,0,0,-5,3,5,5,0,5
MAKE MESH FROM OBJECT 1,1
DELETE OBJECT 1
```

continued on next page

LISTING-46.4

(continued)

Modifying Limb
vertices Directly

```
REM *** Create root object ***
MAKE OBJECT SPHERE 1,0

REM *** and add limb ***
ADD LIMB 1,1,1

REM *** Display current shape till key press ***
WAIT KEY

REM *** Capture triangle's vertex data ***
LOCK VERTEXDATA FOR LIMB 1,1

REM *** Modify vertex in triangle ***
SET VERTEXDATA POSITION 1,0,5,0

REM *** Update screen ***
UNLOCK VERTEXDATA

REM *** End program ***
WAIT KEY
END

FUNCTION SetUpScreen()
    SET DISPLAY MODE 1280,1024,32
    COLOR BACKDROP RGB(255,255,100)
    BACKDROP ON
    AUTOCAM OFF
    POSITION CAMERA 0,0,-20
    POINT CAMERA 0,0,0
ENDFUNCTION
```

Activity 46.5

Type in and test the program in LISTING-46.4 (*vertex04.dbpro*).

Modify the SET VERTEX POSITION statement so that vertex 2 changes coordinates to (7,10,0).

How does this affect the triangle?

Actually, we can modify an object directly without having to convert it to a mesh or a limb. The trick is to remember that the root object in a model is referred to as limb zero. So, by specifying an object's ID number and using a limb number value of zero, we can directly modify normal objects using the LOCK VERTEXDATA FOR LIMB.

For example, if we create a triangle with the statement

```
MAKE OBJECT TRIANGLE 1,0,0,0,-5,3,5,5,0,5
```

we can then capture the triangle's vertex data with the line:

```
LOCK VERTEXDATA FOR LIMB 1,0
```

After that, it is just a matter of the usual vertex manipulation and then unlocking the vertex data buffer.

Activity 46.6

Replace the main section of your last program with the lines

```
SetUpScreen()

REM *** Make triangle ***
MAKE OBJECT TRIANGLE 1,0,0,0,-5,3,5,5,0,5
WAIT KEY
REM *** Capture its vertex data ***
LOCK VERTEXDATA FOR LIMB 1,0
REM *** Modify vertex in triangle ***
SET VERTEXDATA POSITION 2,7,10,0
REM *** Update screen ***
UNLOCK VERTEXDATA
REM *** End program ***
WAIT KEY
END
```

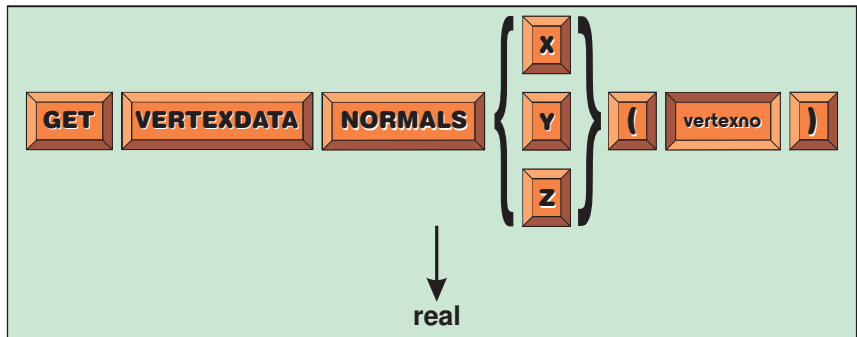
Does the new program create the same effect as the previous version?

The GET VERTEXDATA NORMALS Statement

A vector normal is a line originating from a vertex and is used by the program to calculate lighting effects on a surface near that vertex. A normal has a length of exactly one world unit and the direction in which it points is determined by its end-point (the start point being at the vertex). The coordinates of that end-point can be retrieved using the GET VERTEXDATA NORMALS statement which has the format shown in FIG-46.9.

FIG-46.9

The GET
VERTEXDATA
NORMALS Statement



In the diagram:

`X,Y,Z`

Use one of these options appropriate for the ordinate required.

`vertexno`

is an integer value giving the number of the vertex whose normal's coordinates are to be retrieved.

For example, we could retrieve the triangle's vertex normal at vertex zero using the lines:

```
x# = GET VERTEXDATA NORMALS X(0)
```

```
y# = GET VERTEXDATA NORMALS Y(0)
z# = GET VERTEXDATA NORMALS Z(0)
```

Activity 46.7

Modify your previous program to display the vertex normals for all three points on the triangle using the following logic:

```
FOR vertexno = 0 TO 2
  Get x,y, and z ordinates for vector normal
  Position cursor at 100, 100 + vertexno * 20
  Display x,y,z values
ENDFOR
```

The SET VERTEXDATA NORMALS Statement

It is also possible to modify the vertex normals and hence affect the shading of the surface beside that normal. This is done using the SET VERTEXDATA NORMALS statement which has the format shown in FIG-46.10.

FIG-46.10

The SET VERTEXDATA NORMALS Statement



In the diagram:

vertexno

is an integer value giving the number of the vertex whose vertex normal is to be changed.

x,y,z

are a set of real numbers giving the new coordinates to be assigned to the specified vertex normal.

For example, we could change the normal at vertex 0 in our triangle to (0,0,0) using the line:

```
SET VERTEXDATA NORMALS 0,0,0,0
```

As with any other change, the screen display will only change when the vertex data is unlocked.

In the next program (see LISTING-46.5) we eliminate the normal at each vector by setting their values to (0,0,0) to show how this affects the lighting.

LISTING-46.5

Changing Vertex
Normals

```
SetUpScreen()

MAKE OBJECT TRIANGLE 1,0,0,0,-5,3,5,5,0,5
WAIT KEY
REM ** FOR each vertex normal DO **
FOR v = 0 TO 2
  REM *** eliminate its normal ***
  LOCK VERTEXDATA FOR LIMB 1,0
  SET VERTEXDATA NORMALS v,0,0,0
  UNLOCK VERTEXDATA
  WAIT KEY
NEXT v
```

continued on next page

LISTING-46.5

(continued)

Changing Vertex
Normals

```
REM *** End program ***
WAIT KEY
END

FUNCTION SetUpScreen()
  SET DISPLAY MODE 1280,1024,32
  COLOR BACKDROP RGB(255,255,100)
  BACKDROP ON
  AUTOCAM OFF
  POSITION CAMERA 0,0,-20
  POINT CAMERA 0,0,0
ENDFUNCTION
```

Activity 46.8

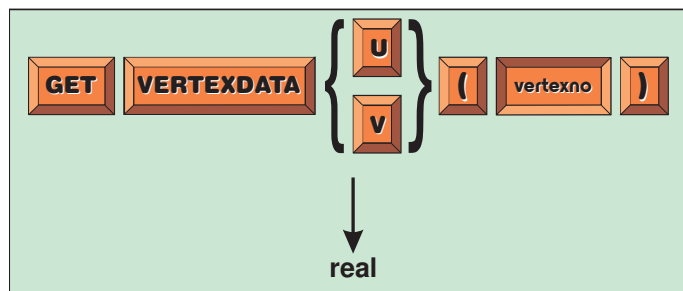
Type in and test the program in LISTING-46.5 (*vertex05.dbpro*).

The GET VERTEXDATA Statement

If a 3D object has been textured, then the UV offset at each vertex can be discovered using the GET VERTEXDATA statement which has the format given in FIG-46.11.

FIG-46.11

The GET
VERTEXDATA UV
Statement



In the diagram:

U,V

Choose the appropriate option for the offset data required.

vertexno

is an integer value specifying the vertex whose UV offsets are to be discovered.

The program in LISTING-46.6 displays the UV offsets for each vertex of a triangle.

LISTING-46.6

Displaying Vertex UV
Settings

```
SetUpScreen()
MAKE OBJECT TRIANGLE 1,0,0,0,-5,3,5,5,0,5
LOAD IMAGE "Spot1.bmp",1
TEXTURE OBJECT 1,1
LOCK VERTEXDATA FOR LIMB 1,0
DO
  FOR vertexno = 0 TO 2
    u# = GET VERTEXDATA U(vertexno)
    v# = GET VERTEXDATA V(vertexno)
    SET CURSOR 100, 100 + vertexno * 20
    PRINT "UV value for vertex ",vertexno," U: ",STR$(u#,2),
      " V : ",STR$(v#,2)
  NEXT v
LOOP
WAIT KEY
END
```

continued on next page

LISTING-46.6
(continued)

Displaying Vertex UV
Settings

```
FUNCTION SetUpScreen()  
  SET DISPLAY MODE 1280,1024,32  
  COLOR BACKDROP 0  
  BACKDROP ON  
  AUTOCAM OFF  
  POSITION CAMERA 0,0,-20  
  POINT CAMERA 0,0,0  
ENDFUNCTION
```

Activity 46.9

Type in and test the program in LISTING-46.6 (*vertex06.dbpro*).

The values displayed are the default settings.

Modify the texturing of the triangle by adding the line

```
    SCROLL OBJECT TEXTURE 1,0.2,0.3
```

immediately after the triangle is textured.

How does this affect the results displayed.

The SET VERTEXDATA UV Statement

To modify the UV offset of an object's texture at a particular vertex, we can use the SET VERTEXDATA UV statement which has the format shown in FIG-46.12.

FIG-46.12

The SET VERTEXDATA
UV Statement



In the diagram:

- | | |
|-----------------|---|
| <i>vertexno</i> | is an integer value specifying the vertex whose UV offsets are to be changed. |
| <i>Uoffset</i> | is a real number giving the offset to be applied along the U axis (range: -1 to 1). |
| <i>Voffset</i> | is a real number giving the offset to be applied along the V axis (range: -1 to 1). |

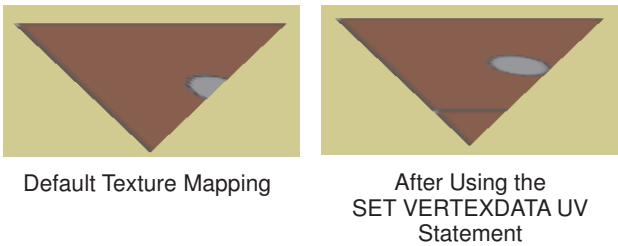
For example, the line

```
SET VERTEXDATA UV 0,-0.4,0.0
```

creates the effect shown in FIG-46.13 on a triangle textured with the white dot on a red background image.

FIG-46.13

The Effects of
Modifying Vertex UV
Settings



The program in LISTING-46.7 scrolls the texture at point zero in a triangle by continually changing the UV offset values.

LISTING-46.7

Modifying Vertex UV
Values

```
SetUpScreen()  
  
REM *** Load texture image ***  
LOAD IMAGE "spot1.bmp",1  
  
REM *** create triangle and texture ***  
MAKE OBJECT TRIANGLE 1,0,0,0,-5,5,0,5,5,0  
TEXTURE OBJECT 1,1  
  
REM *** Capture the vertices ***  
WAIT KEY  
LOCK VERTEXDATA FOR LIMB 1,0,2  
  
REM *** Modify UV offset values for vertex 0 ***  
FOR offset# = -1 TO 1 STEP 0.1  
    SET VERTEXDATA UV 0,offset#,offset#  
    WAIT 100  
NEXT offset#  
  
REM *** End program ***  
WAIT KEY  
END  
  
FUNCTION SetUpScreen()  
    SET DISPLAY MODE 1280,1024,32  
    COLOR BACKDROP RGB(200,200,100)  
    BACKDROP ON  
    AUTOCAM OFF  
    POSITION CAMERA 0,10,-30  
    POINT CAMERA 0,0,0  
ENDFUNCTION
```

Activity 46.10

Type in and test the program given in LISTING-46.7 (*vertex07.dbpro*).

Modify the program to offset vertices 0 and 1 by the value of *offset#*.

The SET VERTEXDATA DIFFUSE Statement

The diffuse setting of a vertex can be modified using the SET VERTEXDATA DIFFUSE statement which has the format shown in FIG-46.14.

FIG-46.14

The SET
VERTEXDATA Diffuse
Statement



In the diagram:

vertexno

is an integer value specifying the vertex whose
diffuse setting is to be changed.

colour

is a integer number (usually created using RGB)
specifying the diffuse colour to be used.

For example, we could create a yellow diffuse at vertex 2 using the line:

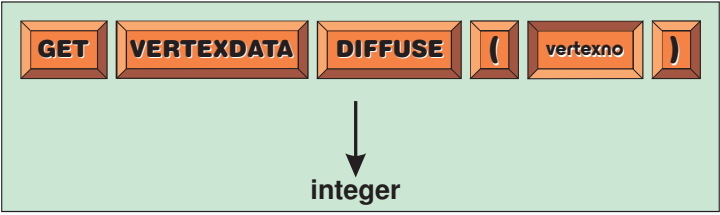
```
SET VERTEXDATA DIFFUSE 2, RGB(255,255,0)
```

The GET VERTEXDATA DIFFUSE Statement

The current diffuse setting of a specified vertex can be discovered using the GET VERTEXDATA DIFFUSE statement which has the format shown in FIG-46.15.

FIG-46.15

The GET
VERTEXDATA
DIFFUSE Statement



vertexno

is an integer value specifying the vertex whose diffuse setting is to be found.

An integer value is returned. This represents the colour diffuse setting.

A typical statement would be:

```
vertcolour = GET VERTEXDATA DIFFUSE (2)
```

Handling More Complex Shapes

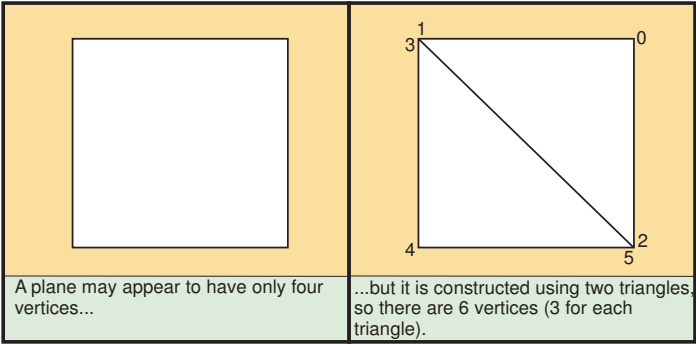
So far we've demonstrated this batch of commands using only a triangle, since that is the simplest shape we can create. But now it's time to look at more complex objects.

Next up is the plane. At first we might be fooled into thinking that a plane can be represented by 4 vertices, but, in fact, 6 are recorded in the vertex buffer.

Two triangles are required to create a plane and each triangle consists of 3 vertices (see FIG-46.16).

FIG-46.16

All Objects are Created
from Triangles



Activity 46.11

How many vertices should be required to define a cube?

The program in LISTING-46.8 displays a plane in wireframe mode and lists the coordinates of each of its vertices.

LISTING-46.8

Displaying Vertex
Coordinates

```
SetUpScreen()
REM *** Create wireframe plane ***
MAKE OBJECT PLAIN 1,10,10
SET OBJECT WIREFRAME 1,1
REM *** Capture vertex data ***
LOCK VERTEXDATA FOR LIMB 1,0
noofvertices = GET VERTEXDATA VERTEX COUNT()
REM *** Display vertex details ***
DO
    SET CURSOR 100,100
    PRINT "Vertices : ",noofvertices
    FOR vertexno = 0 TO noofvertices - 1
        x# = GET VERTEXDATA POSITION X(vertexno)
        y# = GET VERTEXDATA POSITION Y(vertexno)
        z# = GET VERTEXDATA POSITION Z(vertexno)
        SET CURSOR 100, 120 + vertexno * 20
        PRINT vertexno, "    x : ",Format$(x#,3,2),"    y : ",
            Format$(y#,3,2),"    z : ",Format$(z#,3,2),
    NEXT vertexno
LOOP
REM *** End program ***
END

FUNCTION Format$(v#,intpart,decpl)
    result$ = STR$(v#,decpl)
    result$ = SPACE$(intpart+decpl-LEN(result$))+result$
ENDFUNCTION result$

FUNCTION SetUpScreen()
    SET DISPLAY MODE 1280,1024,32
    COLOR BACKDROP 0
    BACKDROP ON
    AUTOCAM OFF
    POSITION CAMERA 0,10,-20
    POINT CAMERA 0,0,0
ENDFUNCTION
```

Activity 46.12

Type in and test the program in LISTING-46.8 (*vertex08.dbpro*).

Change the code so that a cube (10x10x10) is created. Does the number of vertices used match the predicted figure?

We'll discover why a cube does not use the number of vertices we predicted later.

The vertex data specifies coordinates relative to the centre of the object, not as absolute coordinates in 3D space. We can see this from the results of the Activity below.

Activity 46.13

In your previous program (*vertex08.dbpro*) immediately after the wireframe statement, add the line:

```
POSITION OBJECT 1,10,10,0
```

How does this affect the vertex coordinates displayed on the screen?

Another cause for confusion is that the vertex coordinates are measured using reversed x and z axes. This means that negative x values are to the right and negative

z values are "into the screen". For example, the coordinates of the plane's vertex zero are given as (-5,5,0) and so we might be tempted to assume that vertex zero is at the top-left corner. In fact, vertex zero is at the top-right corner, the -5 ordinate being a result of the reversed x-axis measurements. LISTING-46.9 demonstrates this point by incrementally moving vertex zero from (-5,5,0) to (-6,5,0).

LISTING-46.9

Modifying a Vertex's
Coordinates

```
SetUpScreen()

REM *** Create wireframe plane ***
MAKE OBJECT PLAIN 1,10,10
SET OBJECT WIREFRAME 1,1

REM *** Get position of vertex zero ***
WAIT KEY
LOCK VERTEXDATA FOR LIMB 1,0
vertexno = 0
startx# = GET VERTEXDATA POSITION X(vertexno)
y# = GET VERTEXDATA POSITION Y(vertexno)
z# = GET VERTEXDATA POSITION Z(vertexno)

REM *** Assign a new x ord for vertex ***
finx# = -6

REM *** Modify vertex position incrementally ***
FOR c# = startx# TO finx# STEP -0.1
    SET VERTEXDATA POSITION vertexno, c#,y#,z#
    WAIT 100
NEXT c#

REM *** End program ***
WAIT KEY
END

FUNCTION SetUpScreen()
    SET DISPLAY MODE 1280,1024,32
    COLOR BACKDROP 0
    BACKDROP ON
    AUTOCAM OFF
    POSITION CAMERA 0,0,-20
    POINT CAMERA 0,0,0
ENDFUNCTION
```

Activity 46.14

Type in and test the program in LISTING-46.9 (*vertex09.dbpro*).

Activity 46.15

Run *vertex08.dbpro* again.


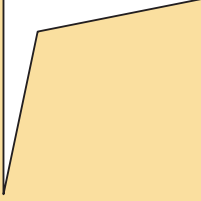
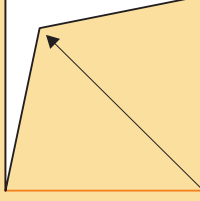
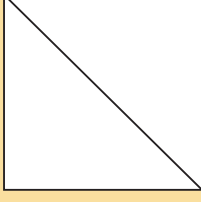
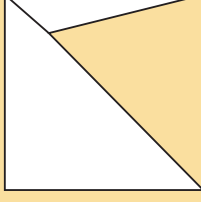
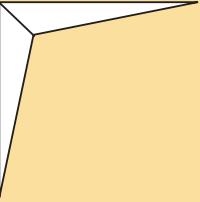
Which vertices have the coordinates (5,5,0)?

Two other vertices have matching coordinates. What are those coordinates?

If we want to modify the position of a vertex, we may have to reposition all the entries in the vertex data area which match that value. For example, in the plane created in the program above.

To move a single vertex of a 3D object it may be necessary to assign the new coordinates to several vertex data entries (see FIG-46.17)

FIG-46.17 Modifying an Plane's Shape

		
If we start with a plane, and wish to distort it to...	... the shape shown above...	... we need to move the bottom-left corner to the new position as shown.
		
Since the original shape is constructed from two triangles...	... adjusting a single entry in the vertex data modifies a single triangle.	To achieve the correct shape, both vertex data entries must be changed.

In a complex 3D shape a single point in space can be repeated over many vertices, so it will be useful if we can find the position of every matching entry in the vertex data.

To do that we can create a function *FindMatchingVertices()*. The routine takes as parameters the object number whose vertices are to be searched and the number of the vertex for which all matches are to be found. Since several vertices may match the one specified, we'll need an array to store the matching vertex numbers. In addition, we need to keep a count of how many matches are actually found:

```
DIM matchvertx(12)
count AS INTEGER
```

Since a function cannot return an array, we'll declare the array to be global. It's probably easiest just to make *count* global too:

```
GLOBAL DIM matchvertx(12)
GLOBAL count AS INTEGER
```

The routine itself has the following logic:

```
Capture vertex data for specified object
Get the coordinates for the specified vertex number
Set count to zero
FOR each vertex DO
    IF the vertex's coords match those of specified vertex THEN
        Add 1 to count
        Add vertex number to matchvertx[count]
    ENDIF
ENDFOR
```

The actual code is :

```

FUNCTION FindMatchingVertices(objno,vertexno)
  REM *** Capture the object's vertex data ***
  LOCK VERTEXDATA FOR LIMB objno,0
  REM *** get coords of specified vertex ***
  x# = GET VERTEXDATA POSITION X(vertexno)
  y# = GET VERTEXDATA POSITION Y(vertexno)
  z# = GET VERTEXDATA POSITION Z(vertexno)

  REM *** Set count to zero ***
  count = 0

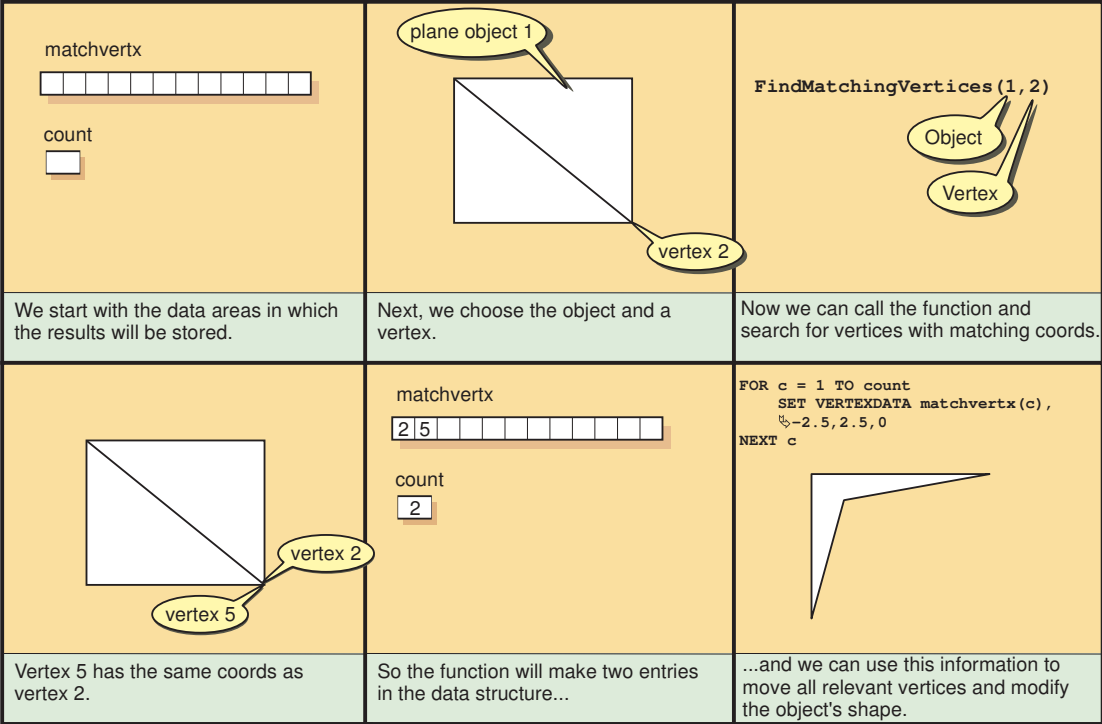
  REM FOR each vertex DO ***
  FOR vno = 0 TO GET VERTEXDATA VERTEX COUNT()-1
    REM *** IF its coords match specified vertex coords ***
    IF x# = GET VERTEXDATA POSITION X(vno) AND
      y# = GET VERTEXDATA POSITION Y(vno) AND
      z# = GET VERTEXDATA POSITION Z(vno)
      REM *** Add to count ***
      INC count
      REM *** Add vertex number to array ***
      matchvertx(count) = vno
    ENDIF
  NEXT vno

  REM *** Restore data ***
  UNLOCK VERTEXDATA
ENDFUNCTION

```

FIG-46.18 demonstrates the effect of searching for matches for vertex 2 of the rectangle used earlier.

FIG-46.18 How to Change all Common Vertices



The program in LISTING-46.10 shows the technique in practice, adjusting the points on a plane to create the arrow shape shown above.

LISTING-46.10

Modifying all Matching
Vertices

```
GLOBAL DIM matchvertx(12)
GLOBAL count AS INTEGER
SetUpScreen()
REM *** Create plane ***
MAKE OBJECT PLAIN 1,10,10
REM *** Find matches for vertex 2
FindMatchingVertices(1,2)
WAIT KEY
REM *** Reposition matching vertices ***
LOCK VERTEXDATA FOR LIMB 1,0
FOR c = 1 TO count
    p = matchvertx(c)
    SET VERTEXDATA POSITION p,3.5,3.5,0
NEXT c
UNLOCK VERTEXDATA
REM *** End program ***
WAIT KEY
END

FUNCTION FindMatchingVertices(objno,vertexno)
    LOCK VERTEXDATA FOR LIMB objno,0
    x# = GET VERTEXDATA POSITION X(vertexno)
    y# = GET VERTEXDATA POSITION Y(vertexno)
    z# = GET VERTEXDATA POSITION Z(vertexno)
    count = 0
    FOR vno = 0 TO GET VERTEXDATA VERTEX COUNT()-1
        IF x# = GET VERTEXDATA POSITION X(vno) AND
            y# = GET VERTEXDATA POSITION Y(vno) AND
            z# = GET VERTEXDATA POSITION Z(vno)
            INC count
            matchvertx(count) = vno
        ENDIF
    NEXT vno
    UNLOCK VERTEXDATA
ENDFUNCTION

FUNCTION SetUpScreen()
    SET DISPLAY MODE 1280,1024,32
    COLOR BACKDROP 0
    BACKDROP ON
    AUTOCAM OFF
    POSITION CAMERA 0,0,-20
    POINT CAMERA 0,0,0
ENDFUNCTION
```

Activity 46.16

Type in and test the program given in LISTING-46.10 (*vertex10.dbpro*).

Modify the program so that the reshaping of the plane is performed incrementally, creating an animated effect.

By adding the line

```
TURN OBJECT LEFT 1,2
```

make the plane rotate about its y-axis while it is changing shape.

With a bit of imagination we can transform basic 3D shapes into more interesting forms. For example, we can create a hemisphere by modifying all the vertices with y values less than zero to have a y-ordinate of zero as shown in LISTING-46.11.

LISTING-46.11

Creating a Hemisphere

```
SetUpScreen()
REM *** Create textured sphere ***
MAKE OBJECT SPHERE 1,10,50,50
LOAD IMAGE "grid8by8.bmp",1
TEXTURE OBJECT 1,1
REM *** Zeroise all y ords < 0 ***
LOCK VERTEXDATA FOR LIMB 1,0
noofvertices = GET VERTEXDATA VERTEX COUNT()
FOR vertexno = 0 TO noofvertices - 1
    y# = GET VERTEXDATA POSITION Y(vertexno)
    IF y# < 0
        x# = GET VERTEXDATA POSITION X(vertexno)
        y# = 0
        z# = GET VERTEXDATA POSITION Z(vertexno)
        SET VERTEXDATA POSITION vertexno,x#,y#,z#
    ENDIF
NEXT vertexno
UNLOCK VERTEXDATA
REM *** Rotate the object ***
ZROTATE OBJECT 1, 90
DO
    PITCH OBJECT UP 1,1
    WAIT 10
LOOP
REM *** End program ***
END

FUNCTION SetUpScreen()
    SET DISPLAY MODE 1280,1024,32
    COLOR BACKDROP 0
    BACKDROP ON
    AUTOCAM OFF
    POSITION CAMERA 0,0,-20
    POINT CAMERA 0,0,0
ENDFUNCTION
```

Activity 46.17

Type in and test the program in LISTING-46.11 (*vertex11.dbpro*).

Modify the line

`y# = 0`

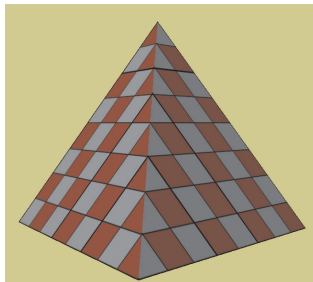
to

`y# = -y#`

How does this affect the shape created?

Activity 46.18

Write a program (act4618.dbpro) which converts a cube to a pyramid (as below) and rotates the final shape.



The ADD MESH TO VERTEXDATA Statement

Should we require it, the vertex details of a second object can be added to the vertex data buffer. However, only data from a mesh can be added, so the object whose details are to be appended to the buffer must first be duplicated in the form of a mesh. For example, if we needed the vertices of a triangle object to be added to an existing vertex data buffer, we would begin by creating the triangle

```
MAKE OBJECT TRIANGLE 2,-1,-1,-1,0,0,0,1,1,1
```

and then creating an equivalent mesh:

```
MAKE MESH FROM OBJECT 1,2
```

With the necessary mesh available, we now use the ADD MESH TO VERTEXDATA statement to add the new vertex details to the buffer. The ADD MESH TO VERTEXDATA statement has the format shown in FIG-46.19.

FIG-46.19

The ADD MESH TO VERTEXDATA Statement



In the diagram:

meshno

is an integer value giving the ID of the mesh whose vertex data is to be added to the buffer.

Using this statement, we can add the details of the triangle object to the buffer with the line:

```
ADD MESH TO VERTEXDATA 1      `Add mesh 1's data to buffer
```

The program in LISTING-46.12 adds a triangle's data to that of a cube in the vertex data buffer, displaying the total number of vertices within the buffer both before and after the triangle is added.

LISTING-46.12

Adding Data to the Vertex Data Buffer

```
SetUpScreen()

REM *** Make cube and place its data in buffer ***
MAKE OBJECT CUBE 1,10
LOCK VERTEXDATA FOR LIMB 1,0

REM *** Display number of vertices in buffer ***
noofvertices = GET VERTEXDATA VERTEX COUNT()
mes$= STR$(noofvertices)
DisplayMessage(mes$,200,200)

REM *** Create triangle and equivalent mesh ***
MAKE OBJECT TRIANGLE 2, -1,-1,-1,0,0,0,1,1,1
MAKE MESH FROM OBJECT 1,2

REM *** Add mesh to buffer ***
ADD MESH TO VERTEXDATA 1

REM *** Display number of vertices in buffer ***
noofvertices = GET VERTEXDATA VERTEX COUNT()
mes$= STR$(noofvertices)
DisplayMessage(mes$,200,200)

REM *** End program ***
WAIT KEY
END
```

continued on next page

LISTING-46.12
(continued)

Adding Data to the
Vertex Data Buffer

```
FUNCTION SetUpScreen()  
    SET DISPLAY MODE 1280,1024,32  
    COLOR BACKDROP 0  
    BACKDROP ON  
    AUTOCAM OFF  
    POSITION CAMERA 0,0,-20  
    POINT CAMERA 0,0,0  
ENDFUNCTION  
  
FUNCTION DisplayMessage(mes$,x,y)  
    now = TIMER()  
    WHILE TIMER() - now < 3000  
        SET CURSOR x,y  
        PRINT mes$  
    ENDWHILE  
ENDFUNCTION
```

Activity 46.19

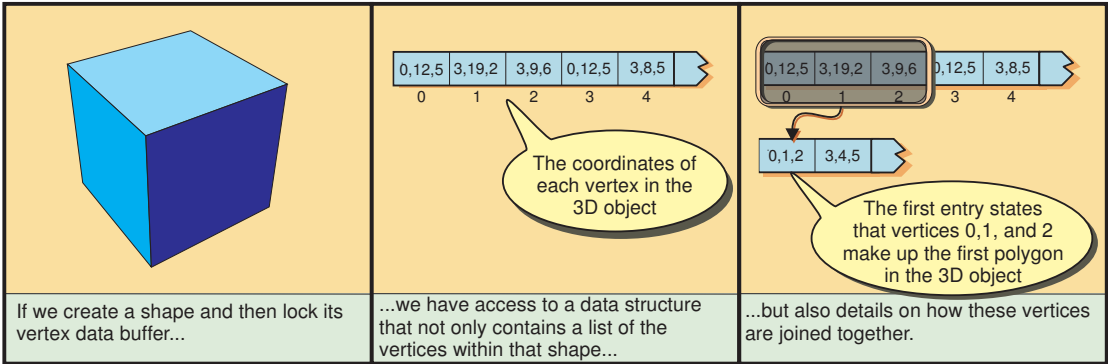
Type in and test the program in LISTING-46.12 (*vertex12.dbpro*).

How many vertices are in the buffer once the triangle's data has been added?

More About the Vertex Data Buffer's Structure

The vertex data buffer is a complex structure; not only does it contain the coordinates of every vertex that makes up a 3D object, but it also contains details of how those vertices are linked to form the polygons that make up that object (see FIG-46.20)

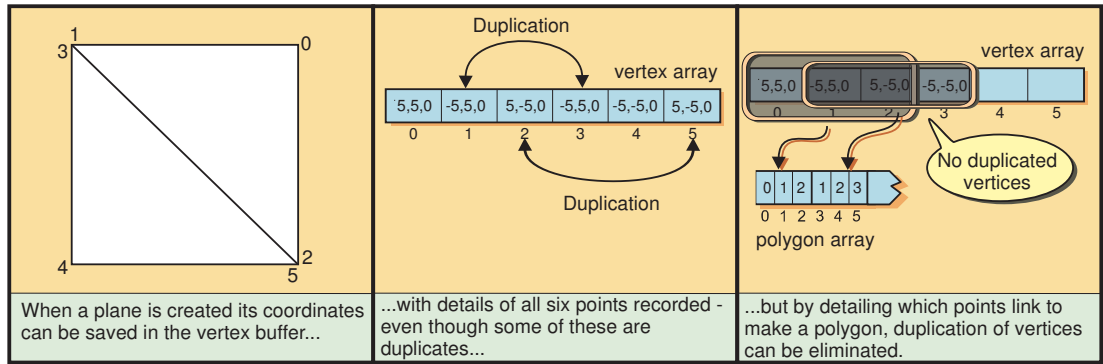
FIG-46.20 Data in the Vertex Buffer



As we can see, a second array within the vertex buffer contains details of how the vertices are linked to form the triangular polygons that make up the structure. The number of entries in this second array should match the number of triangles required to construct the 3D object represented.

For example, a triangle object only needs 1 entry, a plane 2, and a cube 12 (two for each side of the cube). To differentiate between the two sets of data, we'll call the first area the *vertex array* and the second one the *polygon array*. By using this second array detailing the points used to construct each polygon, the need to duplicate vertex coordinates in the first array disappears. The reasoning behind this is shown in FIG-46.21.

FIG-46.21 Alternative Methods of Holding an Object's Details



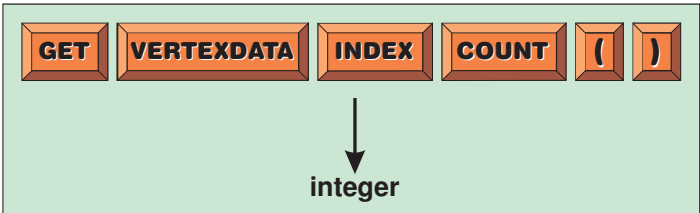
In fact, both methods are used. For simpler objects, such as triangles and planes, only the vertex array is created, but for more complex shapes, such as cubes, duplicate vertices are eliminated from the vertex array and details of which points make up each polygon are recorded in the polygon array.

The GET VERTEXDATA INDEX COUNT Statement

We can discover exactly how many entries are in this polygon array of the vertex buffer using the GET VERTEXDATA INDEX COUNT statement which has the format shown in FIG-46.22.

FIG-46.22

The GET VERTEXDATA INDEX COUNT Statement



The statement actually returns the number of values in the polygon array, and, since there are three values for each polygon, we'll get three times the expected amount

The program in LISTING-46.13 creates a cube and displays the number of entries in the vertex array and the polygons used to construct the object. This last value is arrived at by dividing the value returned by GET VERTEXDATA INDEX COUNT by three.

LISTING-46.13

Displaying a Cube's Vertex Data

```
SetUpScreen()
REM *** Make cube and place its data in buffer ***
MAKE OBJECT CUBE 1,10
LOCK VERTEXDATA FOR LIMB 1,0,2
SYNC
REM *** Display number of vertices and polygons ***
noofvertices = GET VERTEXDATA VERTEX COUNT()
noofpolygons = GET VERTEXDATA INDEX COUNT()/3
PRINT "Vertices      : ", noofvertices
PRINT "Polygons      : ", noofpolygons
SYNC
REM *** End program ***
WAIT KEY
END
```

continued on next page

LISTING-46.13
(continued)

Displaying a Cube's
Vertex Data

```
FUNCTION SetUpScreen()  
  SET DISPLAY MODE 1280,1024,32  
  COLOR BACKDROP 0  
  BACKDROP ON  
  AUTOCAM OFF  
  POSITION CAMERA 0,0,-20  
  POINT CAMERA 0,0,0  
  SYNC ON  
ENDFUNCTION
```

Activity 46.20

Type in and test the program in LISTING-46.13 (*vertex13.dbpro*).

What results are returned when the object created is:

- a) a triangle
- b) a plane
- c) a cone

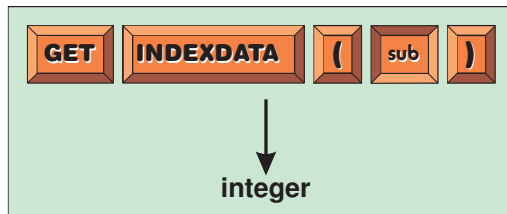
Notice that the triangle and plane return a value of zero for the polygon count. This is because there are only three vertices in the triangle, so retaining information about how these should be joined is unnecessary; the plane, on the other hand, retains duplicated vertices and polygons are created by using the vertex array entries in groups of three (that is, entries 0, 1 and 2 form the first polygon; 3,4 and 5 the second polygon; etc.)

The GET INDEXDATA Statement

We can find out the contents of individual cells within the polygon array using the GET INDEXDATA statement which has the format shown in FIG-46.23.

FIG-46.23

The GET INDEXDATA
Statement



In the diagram:

sub

is an integer value specifying the element of the polygon array to be accessed. The first element is element zero.

We could use this statement to discover which entries in the vertex array are used to construct a specific polygon. For example, assuming we have stored data about a cube within the vertex buffer, we could discover which cells of the vertex array are used in the construction of the first polygon on the first face of the cube using the lines

```
v1 = GETINDEXDATA(0)  
v2 = GETINDEXDATA(1)  
v3 = GETINDEXDATA(2)
```

and with this information we could go on to discover the coordinates of each of the three points:

```

x1 = GET VERTEXDATA POSITION X(v1)
y1 = GET VERTEXDATA POSITION Y(v1)
z1 = GET VERTEXDATA POSITION Z(v1)

```

Activity 46.21

Write a program which creates a 10 unit cube and displays which elements of the vertex array are used to construct its first two polygons.

The program in LISTING-46.14 takes the program in Activity 46.21 a stage further and displays the coordinates used in constructing the second polygon of the cube.

LISTING-46.14

Displaying Vertex
Coordinates

```

TYPE CoordType
  x#
  y#
  z#
ENDTYPE

DIM polycoords(3) AS CoordType

SetUpScreen()

REM *** Make cube and place its data in buffer ***
MAKE OBJECT CUBE 1,10
LOCK VERTEXDATA FOR LIMB 1,0,2
REM *** Collect coordinates ***
FOR c = 1 TO 3
  v = GET INDEXDATA(c+2)
  polycoords(c).x# = GET VERTEXDATA POSITION X(v)
  polycoords(c).y# = GET VERTEXDATA POSITION Y(v)
  polycoords(c).z# = GET VERTEXDATA POSITION Z(v)
NEXT c

REM *** Display the data collected ***
SET CURSOR 0,820
PRINT "Coordinates of second polygon are:"
FOR c = 1 TO 3
  PRINT "(" , polycoords(c).x# , ", " , polycoords(c).y# , ", " ,
    polycoords(c).z# , ")"
NEXT c

REM *** End program ***
WAIT KEY
END

FUNCTION SetUpScreen()
  SET DISPLAY MODE 1280,1024,32
  AUTOCAM OFF
  POSITION CAMERA 0,0,-20
  POINT CAMERA 0,0,0
  SET CAMERA VIEW 0,0,800,600
ENDFUNCTION

```

Activity 46.22

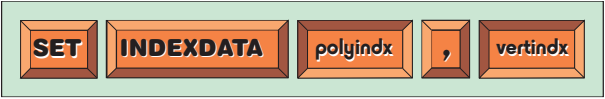
Type in and test the program in LISTING-46.14 (*vertex14.dbpro*).

The SET INDEXDATA Statement

It is possible to modify the contents of the polygon array, thereby changing which three entries in the vertex array are used to construct a polygon. This is achieved using the SET INDEXDATA statement which has the format shown in FIG-46.24.

FIG-46.24

The SET INDEXDATA Statement



In the diagram:

- polyindx*
- is an integer value giving the element in the polygon array which is to have its contents changed. This should lie between 0 and GET VERTEXDATA INDEX COUNT()-1.
- vertindx*
- is an integer value specifying the new value. This will correspond to the index of a cell in the vertex array. This should lie between 0 and GET VERTEXDATA VERTEX COUNT()-1.

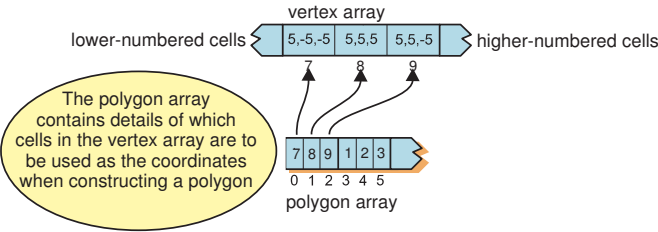
For example, assuming a cube's details are stored in the vertex buffer, we could change the vertices used to construct the first polygon in the cube to those stored in cells 7, 8, and 9 of the vertex array using the lines:

```
SET INDEXDATA 0, 7
SET INDEXDATA 1, 8
SET INDEXDATA 2, 9
```

FIG-46.25 gives a visual interpretation of the operation.

FIG-46.25

The Result of Using SET INDEXDATA



In LISTING-46.15 a cube has a vertex specification belonging to its first polygon changed to reference a different cell in the vertex array. The modified cube is rotated to show how this change has affected its appearance.

LISTING-46.15

Modifying a Cube's Structure

```
SetUpScreen ()

REM *** Make cube and place its data in buffer ***
MAKE OBJECT CUBE 1,10
LOCK VERTEXDATA FOR LIMB 1,0,2

REM *** Modify details of how the first polygon is constructed ***
WAIT KEY
SET INDEXDATA 0,3

REM *** Switch on culling for more detail ***
SET OBJECT CULL 1,0
```

continued on next page

LISTING-46.15
(continued)

Modifying a Cube's
Structure

```
REM *** Update screen ***
WAIT KEY
UNLOCK VERTEXDATA
REM *** Rotate cube ***
DO
    TURN OBJECT LEFT 1,1
    WAIT 10
LOOP
REM *** End program ***
WAIT KEY
END

FUNCTION SetUpScreen()
    SET DISPLAY MODE 1280,1024,32
    AUTOCAM OFF
    POSITION CAMERA 0,0,-20
    POINT CAMERA 0,0,0
ENDFUNCTION
```

Activity 46.23

Type in and test the program in LISTING-46.15 (*vertex15.dbpro*).

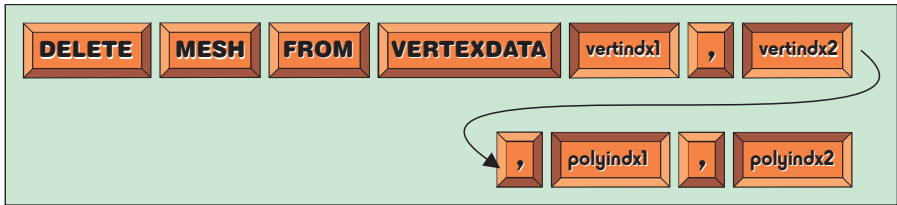
The DELETE MESH FROM VERTEXDATA Statement

Using ADD MESH TO VERTEXDATA we saw that it was possible to add new models to the data in the vertex data buffer. These additions can also be removed from the vertex buffer, but this is a more complex operation since it requires details of which entries in the vertex array and the polygon array have to be deleted.

To delete part of the vertex data buffer we use the DELETE MESH FROM VERTEXDATA statement which has the format shown in FIG-46.26.

FIG-46.26

The DELETE MESH
FROM VERTEXDATA
Statement



In the diagram:

vertindx1,vertindx2

are integer values representing the subscripts
of the first and last cells in the vertex array
whose contents are to be deleted.

polyindx1,polyindx2

are integer values representing the subscripts
of the first and last cells in the polygon array
whose contents are to be deleted.

For example, if we place the vertex data for a cube into the vertex buffer, 24 cells in the vertex array will be occupied (cells 0 to 23) and 36 cells in the polygon array (cells 0 to 35). Adding a second cube to the vertex buffer would result in an additional 24 entries in the vertex array (cells 24 to 47) and 36 extra entries in the polygon array (cells 36 to 71). To delete this second cube from the vertex buffer we would use the statement:

```
DELETE MESH FROM VERTEXDATA 24,47,36,71
```

Summary

- Every 3D Object is constructed using triangles.
- By modifying the coordinates of a vertex, a 3D object can be reshaped.
- To modify an object's vertices, they must first be captured in a buffer area.
- Use `LOCK VERTEXDATA FOR MESH` to capture into the buffer area the vertices that make up a mesh.
- Use `GET VERTEXDATA VERTEX COUNT` to discover how many vertices are stored in the buffer area.
- Use `GET VERTEXDATA POSITION` to access the coordinates of a given vertex.
- Use `SET VERTEXDATA POSITION` to modify the coordinates of a specific vertex.
- Use `UNLOCK VERTEXDATA` to allow the modified data to be applied to the object or mesh to which it is associated.
- Use `LOCK VERTEXDATA FOR LIMB` to capture the vertex data of a limb or object.
- Use `GET VERTEXDATA NORMALS` to access the end coordinates of a given vertex normal.
- Use `SET VERTEXDATA NORMALS` to modify the coordinates of a specific vertex normal.
- Use `GET VERTEXDATA {UV}` to access the U and V offsets at a given vertex for a textured object.
- Use `SET VERTEXDATA UV` to modify the U/V offsets of a specific vertex.
- Use `SET VERTEXDATA DIFFUSE` to modify the diffuse setting for a vertex.
- Use `GET VERTEXDATA DIFFUSE` to access the current diffuse setting for a specific vertex.
- If a vertex's coordinates are modified, generally, all other vertices with identical coordinates should also be modified.
- Manipulation of vertices allows us to create new shapes from the basic 3D primitives. For example, a hemisphere from a sphere or a pyramid from a cube.
- Vertex coordinates are relative to the object's centre and not absolute world coordinates.
- Use `ADD MESH TO VERTEXDATA` to add the details of a mesh to the data already stored in the vertex buffer.
- The vertex data buffer contains an array (the vertex array) which holds the coordinates of every vertex used to make up the 3D object(s) held in the buffer.

- For some simple objects such as a plane, vertex coordinates may be repeated and vertices are used in groups of three to recreate the triangles that make up the original object.
- For more complex shapes, each vertex's coordinates are stored only once and a second array (the polygon array) holds links to those vertices used to construct each polygon.
- Use `GET VERTEXDATA INDEX COUNT` to discover the number of entries in the polygon array.
- Use `GET INDEXDATA` to discover which vertex an element of the polygon array references.
- Use `SET INDEXDATA` to modify which vertex an entry in the polygon array references.
- Use `DELETE MESH FROM VERTEXDATA` to remove a mesh's data from the vertex data buffer.

Solutions

Activity 46.1

No solution required.

Activity 46.2

```
SetUpScreen()

REM *** Create triangle ***
MAKE OBJECT TRIANGLE 1,0,0,0,-5,0,5,5,0,5
REM *** Convert it to a mesh ***
MAKE MESH FROM OBJECT 1,1
DELETE OBJECT 1
REM *** Capture mesh vertex data ***
LOCK VERTEXDATA FOR MESH 1
REM *** Find out no. of vertices in mesh ***
count = GET VERTEXDATA VERTEX COUNT()
REM *** Get coordinates of second vertex ***
x# = GET VERTEXDATA POSITION X(1)
y# = GET VERTEXDATA POSITION Y(1)
z# = GET VERTEXDATA POSITION Z(1)
REM *** Show number of vertices ***
REM *** & vertex 1's coords ***
DO
    SET CURSOR 100,100
    PRINT count
    SET CURSOR 100,120
    PRINT STR$(x#,0), " ", STR$(y#,0), " ",
    ↵ STR$(z#,0)
LOOP
REM *** End program ***
END

FUNCTION SetUpScreen()
    SET DISPLAY MODE 1280,1024,32
    COLOR BACKDROP RGB(255,255,100)
    BACKDROP ON
    AUTOCAM OFF
ENDFUNCTION
```

Activity 46.3

No solution required.

Activity 46.4

No solution required.

Activity 46.5

```
SetUpScreen()
REM *** Make triangular mesh ***
MAKE OBJECT TRIANGLE 1,0,0,0,-5,3,5,5,0,5
MAKE MESH FROM OBJECT 1,1
DELETE OBJECT 1
REM *** Create root object ***
MAKE OBJECT SPHERE 1,0
REM *** and add limb ***
ADD LIMB 1,1,1
WAIT KEY
REM *** Capture triangle's vertex data ***
LOCK VERTEXDATA FOR LIMB 1,1
REM *** Modify vertex in triangle ***
SET VERTEXDATA POSITION 2,7,10,0
REM *** Update screen ***
UNLOCK VERTEXDATA
REM *** End program ***
WAIT KEY
END
```

```
FUNCTION SetUpScreen()
    SET DISPLAY MODE 1280,1024,32
    COLOR BACKDROP RGB(255,255,100)
    BACKDROP ON
    AUTOCAM OFF
    POSITION CAMERA 0,0,-20
    POINT CAMERA 0,0,0
ENDFUNCTION
```

The top right corner of the triangle is moved.

Activity 46.6

The program has the same effect as the previous one.

Activity 46.7

```
SetUpScreen()
REM *** Make triangle ***
MAKE OBJECT TRIANGLE 1,0,0,0,-5,3,5,5,0,5
WAIT KEY
REM *** Capture its vertex data ***
LOCK VERTEXDATA FOR LIMB 1,0
REM *** Modify vertex in triangle ***
SET VERTEXDATA POSITION 2,7,10,0
REM *** Update screen ***
UNLOCK VERTEXDATA
LOCK VERTEXDATA FOR LIMB 1,0
DO
    FOR vertexno = 0 TO 2
        normx# =
            ↵ GET VERTEXDATA NORMALS X(vertexno)
        normy# =
            ↵ GET VERTEXDATA NORMALS Y(vertexno)
        normz# =
            ↵ GET VERTEXDATA NORMALS Z(vertexno)
        SET CURSOR 100, 100 + vertexno * 20
        PRINT normx#, " ", normy#, " ", normz#
    NEXT vertexno
LOOP
REM *** End program ***
WAIT KEY
END

FUNCTION SetUpScreen()
    SET DISPLAY MODE 1280,1024,32
    COLOR BACKDROP RGB(255,255,100)
    BACKDROP ON
    AUTOCAM OFF
    POSITION CAMERA 0,0,-20
    POINT CAMERA 0,0,0
ENDFUNCTION
```

Activity 46.8

No solution required.

Activity 46.9

```
SetUpScreen()
MAKE OBJECT TRIANGLE 1,0,0,0,-5,0,5,5,0,5
LOAD IMAGE "Spot1.bmp",1
TEXTURE OBJECT 1,1
SCROLL OBJECT TEXTURE 1,0.2,0.3
DO
    LOCK VERTEXDATA FOR LIMB 1,0
    FOR vertexno = 0 TO 2
        u# = GET VERTEXDATA U(vertexno)
        v# = GET VERTEXDATA V(vertexno)
```

```

        SET CURSOR 100, 100 + vertexno * 20
        PRINT "UV value for vertex ",
        ↵vertexno, " U: ", STR$(u#,2), " V : ",
        ↵STR$(v#,2)
    NEXT v
UNLOCK VERTEXDATA
LOOP
WAIT KEY
END

FUNCTION SetUpScreen()
    SET DISPLAY MODE 1280,1024,32
    COLOR BACKDROP 0
    BACKDROP ON
    AUTOCAM OFF
    POSITION CAMERA 0,10,-20
    POINT CAMERA 0,0,0
ENDFUNCTION

```

The UV values should now read as:

```

UV value for vertex 0: 0.20  0.30
UV value for vertex 1: 1.20  0.30
UV value for vertex 2: 1.20  1.30

```

Activity 46.10

```

SetUpScreen()
REM *** Load texture image ***
LOAD IMAGE "spot1.bmp",1
REM *** create triangle and texture ***
MAKE OBJECT TRIANGLE 1,0,0,0,-5,5,0,5,5,0
TEXTURE OBJECT 1,1
REM *** Capture the vertices ***
WAIT KEY
LOCK VERTEXDATA FOR LIMB 1,0,2
REM *** Modify UV offset values for vertex
0 ***
FOR offset# = -1 TO 1 STEP 0.1
    SET VERTEXDATA UV 0,offset#,offset#
    SET VERTEXDATA UV 1,offset#,offset#
    WAIT 100
NEXT offset#
REM *** End program ***
WAIT KEY
END

FUNCTION SetUpScreen()
    SET DISPLAY MODE 1280,1024,32
    COLOR BACKDROP RGB(200,200,100)
    BACKDROP ON
    AUTOCAM OFF
    POSITION CAMERA 0,0,-30
    POINT CAMERA 0,0,0
ENDFUNCTION

```

Activity 46.11

We would expect a cube to use 36 vertices. Since a plane requires 6 vertices and there are 6 planes in a cube, we should need a total of 6 * 6 vertices.

Activity 46.12

To make a cube, change the line

```
MAKE OBJECT PLAIN 1,10,10
```

to

```
MAKE OBJECT CUBE 1,10
```

The cube used 24 vertices rather than the predicted 36.

Activity 46.13

The vertex coordinates are unchanged. This means that the coordinates are measured from the centre of the object and not from the world origin.

Activity 46.14

No solution required.

Activity 46.15

Vertices 1 and 3 have the coordinates (5,5,0).

Vertices 2 and 4 have the coordinates (-5,-5,0).

Activity 46.16

Version 1 (animated transform)

```

GLOBAL DIM matchvertx(12)
GLOBAL count AS INTEGER
SetUpScreen()
REM *** Create plane ***
MAKE OBJECT PLAIN 1,10,10
REM *** Find matches for vertex 2 ***
FindMatchingVertices(1,2)
WAIT KEY
REM *** Reposition matching vertices ***
FOR post# = -5 TO 3.5 STEP 0.1
    LOCK VERTEXDATA FOR LIMB 1,0
    FOR c = 1 TO count
        p = matchvertx(c)
        SET VERTEXDATA POSITION p,post#,post#,0
    NEXT c
    UNLOCK VERTEXDATA
    WAIT 50
NEXT post#
REM *** End program ***
WAIT KEY
END

FUNCTION FindMatchingVertices(objno,vertexno)
    LOCK VERTEXDATA FOR LIMB objno,0
    x# = GET VERTEXDATA POSITION X(vertexno)
    y# = GET VERTEXDATA POSITION Y(vertexno)
    z# = GET VERTEXDATA POSITION Z(vertexno)
    count = 0
    FOR vno = 0 TO
        ↵GET VERTEXDATA VERTEX COUNT()-1
        IF x# = GET VERTEXDATA POSITION X(vno)
            ↵AND y#=GET VERTEXDATA POSITION Y(vno)
            ↵AND z#=GET VERTEXDATA POSITION Z(vno)
            INC count
            matchvertx(count) = vno
        ENDIF
    NEXT vno
    UNLOCK VERTEXDATA
ENDFUNCTION

```

```

FUNCTION SetUpScreen()
    SET DISPLAY MODE 1280,1024,32
    COLOR BACKDROP 0
    BACKDROP ON
    AUTOCAM OFF
    POSITION CAMERA 0,0,-20
    POINT CAMERA 0,0,0
ENDFUNCTION

```

Version 2 (rotating animated transform)

```

GLOBAL DIM matchvertx(12)
GLOBAL count AS INTEGER

```

```

SetUpScreen()
REM *** Create plane ***
MAKE OBJECT PLAIN 1,10,10
REM *** Find matches for vertex 2 ***
FindMatchingVertices(1,2)
WAIT KEY
REM *** Reposition matching vertices ***
FOR post# = -5 TO 3.5 STEP 0.1
  LOCK VERTEXDATA FOR LIMB 1,0
  FOR c = 1 TO count
    p = matchvertx(c)
    SET VERTEXDATA POSITION p,post#,post#,0
  NEXT c
  UNLOCK VERTEXDATA
  WAIT 50
NEXT post#
REM *** End program ***
WAIT KEY
END

FUNCTION FindMatchingVertices(objno,vertexno)
  LOCK VERTEXDATA FOR LIMB objno,0
  x# = GET VERTEXDATA POSITION X(vertexno)
  y# = GET VERTEXDATA POSITION Y(vertexno)
  z# = GET VERTEXDATA POSITION Z(vertexno)
  count = 0
  FOR vno = 0 TO
    ↵GET VERTEXDATA VERTEX COUNT()-1
    IF x# = GET VERTEXDATA POSITION X(vno)
      ↵AND y#=GET VERTEXDATA POSITION Y(vno)
      ↵AND z#=GET VERTEXDATA POSITION Z(vno)
      INC count
      matchvertx(count) = vno
    ENDIF
  NEXT vno
  UNLOCK VERTEXDATA
ENDFUNCTION

FUNCTION SetUpScreen()
  SET DISPLAY MODE 1280,1024,32
  COLOR BACKDROP 0
  BACKDROP ON
  AUTOCAM OFF
  POSITION CAMERA 0,0,-20
  POINT CAMERA 0,0,0
ENDFUNCTION

```

Activity 46.17

The program now creates a hollow bowl shape as shown below.



Activity 46.18

```

SetUpScreen()
REM *** Create textured cube ***
MAKE OBJECT CUBE 1,10
LOCK VERTEXDATA FOR LIMB 1,0
LOAD IMAGE "grid8by8.bmp",1
TEXTURE OBJECT 1,1
REM *** Modify vertices ***
noofvertices = GET VERTEXDATA VERTEX
COUNT()
FOR vertexno = 0 TO noofvertices - 1
  x# = GET VERTEXDATA POSITION X(vertexno)
  y# = GET VERTEXDATA POSITION Y(vertexno)
  z# = GET VERTEXDATA POSITION Z(vertexno)
  IF x# <> 0 AND Y# > 0
    x# = 0
    z# = 0
    SET VERTEXDATA POSITION vertexno,
    ↵x#,y#,z#
  ENDIF
NEXT vertexno
UNLOCK VERTEXDATA
REM *** Rotate pyramid ***
DO
  TURN OBJECT LEFT 1,1
  WAIT 10
LOOP

REM *** End program ***
END

FUNCTION SetUpScreen()
  SET DISPLAY MODE 1280,1024,32
  COLOR BACKDROP RGB(200,200,100)
  BACKDROP ON
  AUTOCAM OFF
  POSITION CAMERA 0,0,-20
  POINT CAMERA 0,0,0
ENDFUNCTION

```

Activity 46.19

The cube creates 24 entries in the buffer while the triangle adds another 3 giving a total of 27.

Activity 46.20

The original MAKE OBJECT CUBE statement needs to be changed for each shape we require.

The triangle requires a line such as:

```
MAKE OBJECT TRIANGLE 1,-5,-5,-5,0,0,0,5,5,5
```

This gives 3 vertices and 0 polygon entries.

The plane:

```
MAKE OBJECT PLAIN 1, 10,10
```

vertices: 6 polygon links: 0

The cone:

```
MAKE OBJECT CONE 1, 10
```

vertices: 23 polygon links: 11

Activity 46.21

```
SetUpScreen()
REM *** Make cube and capture its data ***
MAKE OBJECT CUBE 1, 10
LOCK VERTEXDATA FOR LIMB 1,0,2
SYNC
REM *** Display first 2 polygons ***
FOR poly = 0 TO 1
  PRINT "Polygon ",poly," : ";
  FOR vert = 0 TO 2
    PRINT GET INDEXDATA(poly*3+vert)," ";
  NEXT vert
  PRINT
NEXT poly
SYNC
REM *** End program ***
WAIT KEY
END

FUNCTION SetUpScreen()
  SET DISPLAY MODE 1280,1024,32
  COLOR BACKDROP 0
  BACKDROP ON
  AUTOCAM OFF
  POSITION CAMERA 0,0,-20
  POINT CAMERA 0,0,0
  SYNC ON
ENDFUNCTION
```

Activity 46.22

No solution required.

Activity 46.23

No solution required.

Accessing Memory

Bitmap and Image Memblocks

Data Memblocks

Manipulating Memblocks

Mesh Memblocks

Saving and Loading Memblocks

Sound Memblocks

Using Pointers

Accessing Memory

Introduction

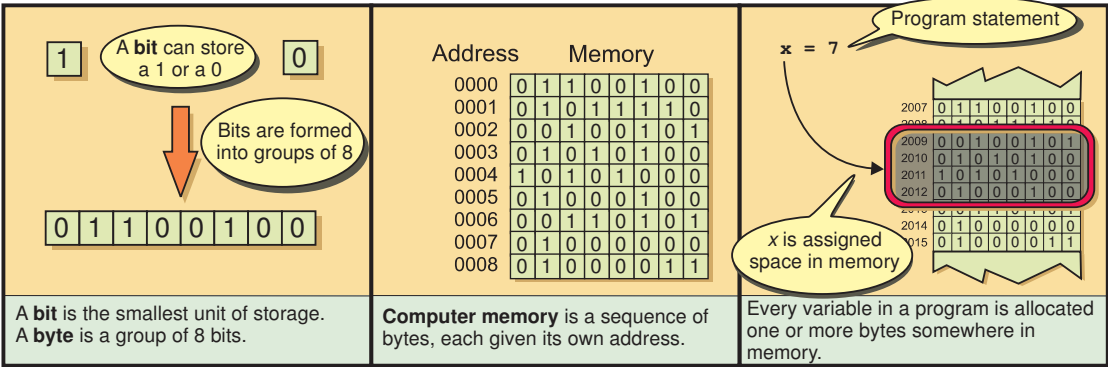
The basic storage unit in computer memory is the **bit**. A bit can store a single binary digit (0 or 1). In a computer's main memory these bits are formed into groups of 8 known as **bytes**. Every byte is allocated an unique memory address. These addresses are allocated in sequential order. Hence the first location is given the address 0, the next address 1, etc.

When we create a variable with a statement such as

```
lives = 3
```

the computer reserves a few bytes of memory for that variable (see FIG-47.1) and it's in those bytes that the variable's contents are stored.

FIG-47.1 Computer Memory



Most of the time, finding out which locations have been allocated to a variable is of no interest to us since we can access its contents by using the variable's name.

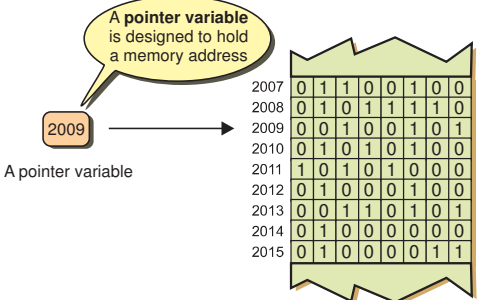
In fact it's not just variables that are allocated memory space; every item of data used by a program, from simple integers to images, sounds and 3D objects will all be allocated memory within RAM when a program is executing.

Pointers

While most variables store the traditional numeric or string values, there is a special variable type capable of storing the address at which an item of normal data is held. Such a variable is known as a **pointer** (see FIG-47.2).

FIG-47.2

A Pointer Variable



Creating Pointers in DarkBASIC Pro

If we intend to use a variable to store a memory address, then it needs to be created as a DWORD variable:

```
ptr AS DWORD
```

Assigning a Value to a Pointer

In theory we can place any integer value in a pointer

```
ptr = 2009
```

and treat that value as a memory address (after all, there's bound to be a memory location which has that address) but, in practice, this won't work since the operating system will not allow a program to access parts of the memory that have not been allocated to that program.

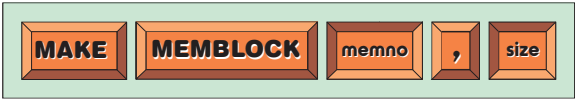
Instead, we need to ask the program to reserve a new area of memory and then place the address of this new area in the pointer.

The MAKE MEMBLOCK Statement

To allocate a new block of memory we need to use the MAKE MEMBLOCK statement which has the format shown in FIG-47.3.

FIG-47.3

The MAKE MEMBLOCK Statement



In the diagram:

- memno* is an integer value specifying the ID to be assigned to the memory block being allocated.
- size* is an integer value giving the number of bytes required in the memory block.

For example we could allocate a single byte to memory block 1 using the line:

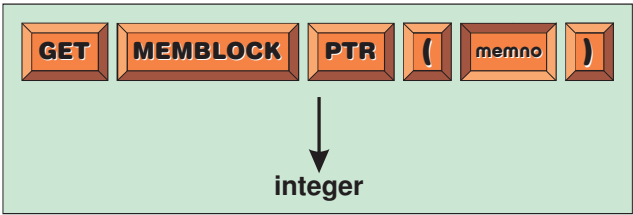
```
MAKE MEMBLOCK 1,1
```

The GET MEMBLOCK PTR Statement

Now we need to acquire the address of the newly allocated memory. This is done using the GET MEMBLOCK PTR statement which has the format shown in FIG-47.4.

FIG-47.4

The GET MEMBLOCK PTR Statement



In the diagram:

memno

is an integer value specifying the ID of the memory block whose address is to be found.

Using this statement, we can assign the memory block's address to our pointer:

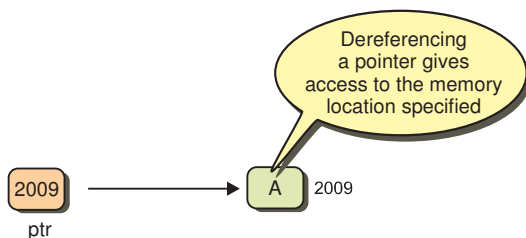
```
ptr = GET MEMBLOCK PTR(1)
```

Using a Pointer

Now that the pointer contains the address of the memory block, we can use the pointer to access the reserved area. To do this we **dereference** the pointer. This simply means that we use the contents of the pointer to access the memory location specified (see FIG-47.5).

FIG-47.5

Dereferencing a Pointer



If this seems a strange idea, imagine looking at a treasure map; the map does not contain the treasure - but it tells you where to find it. A pointer variable doesn't contain the data you're searching for, but it tells you where to find it.

In DarkBASIC Pro we put an asterisk (*) in front of the pointer variable in order to dereference it. For example, the statement

```
*ptr = 65
```

will store the value 65 at the memory location referenced by *ptr*, and

```
PRINT *ptr
```

will display the value held at that location.

The program in LISTING-47.1 brings together everything we've looked at so far, setting up a byte of memory, storing the value 65 at the location and then displaying its contents.

LISTING-47.1

Using a Pointer

```
REM *** Declare the pointer variable ***
ptr AS DWORD

REM *** Reserve a single byte ***
MAKE MEMBLOCK 1,4
REM *** Copy the address of the byte to the pointer ***
ptr = GET MEMBLOCK PTR(1)

REM *** Store a value in the memory location ***
*ptr = 65

REM *** Display the contents of the memory location ***
PRINT (*ptr)
REM *** End program ***
WAIT KEY
END
```

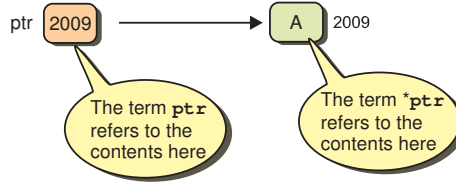
Activity 47.1

Type in and test the program in LISTING-47.1 (*mem01.dbpro*).

It is important that you understand that the term *ptr* refers to the contents of the variable *ptr* and that the term **ptr* refers to the contents of the address specified by *ptr* (see FIG-47.6).

FIG-47.6

ptr and ***ptr**



Using a Pointer to Return Values from a Function

By using a pointer as a parameter to a function we can modify the contents of a memory block from within the function. The program in LISTING-47.2 creates 3 separate memory block areas, each capable of storing a single integer value. The areas are then assigned values: 1, 2 and 3 respectively. The user can then choose which of the three areas are to be updated by the *AddTwo()* function which adds 2 to the area specified in its parameter.

LISTING-47.2

Pointers and Parameters

```
REM *** Declare the pointer variable ***
ptr AS DWORD

REM *** Reserve 3 4-byte areas ***
MAKE MEMBLOCK 1,4
MAKE MEMBLOCK 2,4
MAKE MEMBLOCK 3,4

REM *** Store a value in each area ***
FOR c = 1 TO 3
    ptr = GET MEMBLOCK PTR(c)
    REM *** Store a value in each memory location ***
    *ptr = c
NEXT c

REM *** Get user's choice ***
INPUT "Which area should be updated (1,2,or 3)",area

REM *** Get the address of the selected area ***
ptr = GET MEMBLOCK PTR(area)

REM *** Use function to add 2 to area ***
AddTwo(ptr)

REM *** Display the contents of the area ***
PRINT "That area now contains the value ",(*ptr)

REM *** End program ***
WAIT KEY
END

REM *** Add 2 to specified address ***
FUNCTION AddTwo(p AS DWORD)
    *p = (*p)+2
ENDFUNCTION.
```

To keep the code to a minimum, the input value is not validated.

Activity 47.2

Type in and test the program in LISTING-47.2 (*mem02.dbpro*).

Make sure that it works for each memory area.

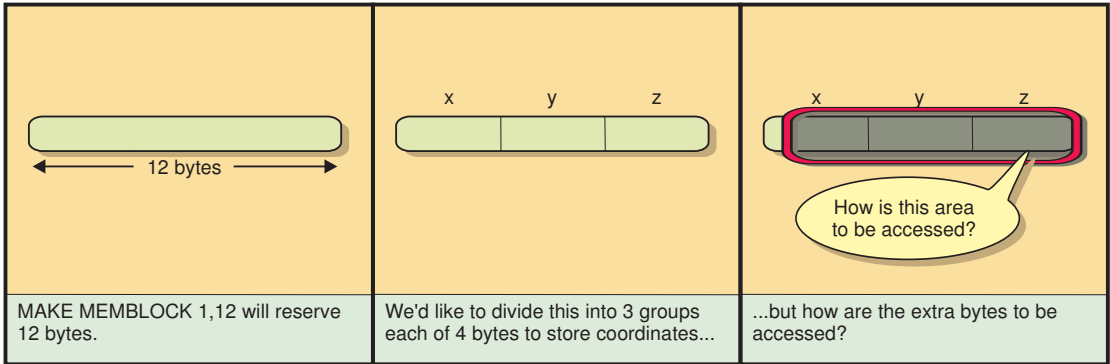
Larger Memory Blocks

We can reserve as large a memory block as we want (within reason). For example, let's say we want to store the current position of a spaceship. That will require space for three real variables (*x*, *y*, *z*). Since a real number occupies 4 bytes, we'll need to reserve a total area of 12 bytes:

```
MAKE MEMBLOCK 1,12
```

The problem now is that we need some method of storing data in the different parts of this area (see FIG-47.7).

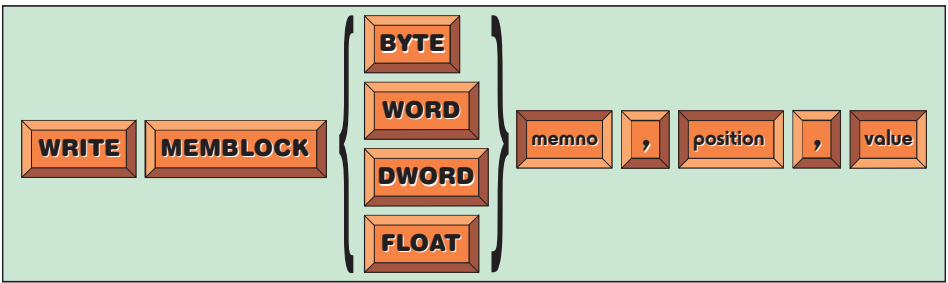
FIG-47.7 How Are Different Areas of a Memory Block Accessed?



The WRITE MEMBLOCK Statement

To write to part of a memory block we must use the WRITE MEMBLOCK statement which allows us to specify exactly where within a memory block data is to be written. The statement has the format shown in FIG-47.8.

FIG-47.8 The WRITE MEMBLOCK Statement



In the diagram:

- | | |
|------|---|
| BYTE | Use this option if a single byte is to be written to the memory block. |
| WORD | Use this option if an integer stored over 4 bytes is to be written to the memory block. |

DWORD	Use this option if an integer stored over 8 bytes is to be written.
FLOAT	Use this option if a real number is to be written.
<i>memno</i>	is an integer value giving the ID of the memory block to which the data is to be written.
<i>position</i>	is an integer value specifying the position of the first byte within the memory block to be used to store the value. When the value is to be stored at the start of the memory block, use zero.
<i>value</i>	is the value to be stored in the memory block. The value's type should match that used earlier in the statement.

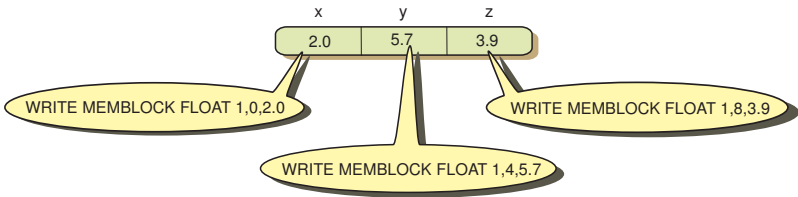
For example, assuming the spaceship's coordinates are (2.0,5.7,3.9), we could write these values to memory block 1 using the statements:

```
WRITE MEMBLOCK FLOAT 1,0,2.0
WRITE MEMBLOCK FLOAT 1,4,5.7
WRITE MEMBLOCK FLOAT 1,8,3.9
```

FIG-47.9 gives a visual representation of the effect of these statements.

FIG-47.9

Storing Multiple Data Values in a Memory Block

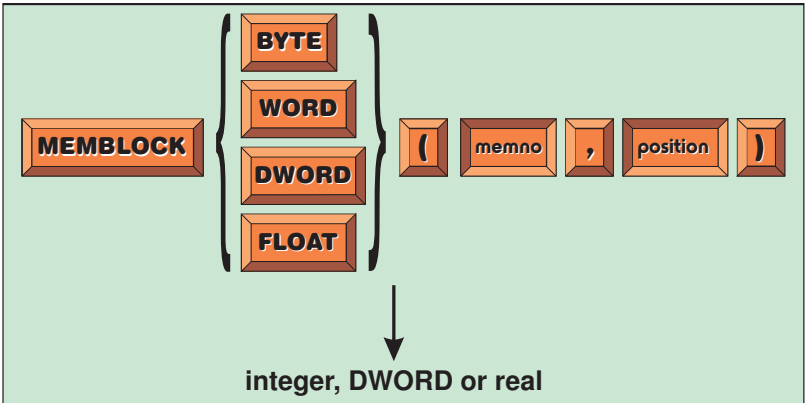


The MEMBLOCK Statement

To access information already stored in a memory block, we use the MEMBLOCK statement which has the format shown in FIG-47.10.

FIG-47.10

The MEMBLOCK Statement



In the diagram:

- BYTE, WORD, DWORD, FLOAT
 - Use the option appropriate for the type of value

being retrieved from the memory area.

memno

is an integer value specifying the ID of the memory block to be accessed.

position

is an integer value specifying where within the memory block the value to be accessed begins.

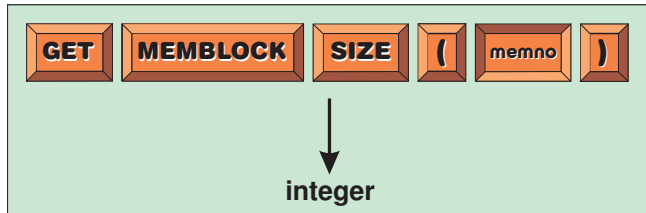
The type of value returned by this statement depends on the type of value being read. The BYTE and WORD options both return an integer value, DWORD returns a DWORD type, and FLOAT returns a real.

The GET MEMBLOCK SIZE Statement

We can discover how many bytes are in a memory block using the GET MEMBLOCK SIZE statement which has the format shown in FIG-47.11.

FIG-47.11

The GET MEMBLOCK
SIZE Statement



In the diagram:

memno

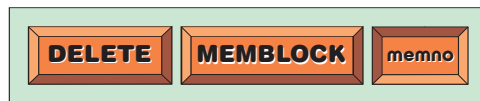
is an integer value specifying the ID of the memory block to be accessed.

The DELETE MEMBLOCK Statement

When it is no longer required, it is best to free up the RAM space allocated to a memory block. To achieve this we can use the DELETE MEMBLOCK statement which has the format shown in FIG-47.12.

FIG-47.12

The DELETE
MEMBLOCK Statement



In the diagram:

memno

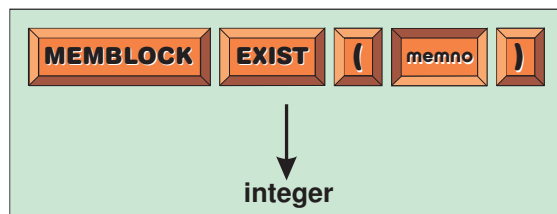
is an integer value specifying the ID of the memory block to be deleted.

The MEMBLOCK EXIST Statement

We can check if a memory block of a specific ID exists using the MEMBLOCK EXIST statement which has the format shown in FIG-47.13.

FIG-47.13

The MEMBLOCK
EXIST Statement



In the diagram:

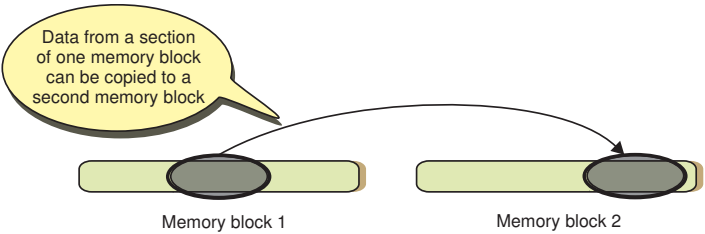
memno is an integer value giving the ID of the memory block to be checked.

The COPY MEMBLOCK Statement

If we set up more than one memory area, then we can copy information from a section of one memory block to some part of the second block (see FIG-47.14).

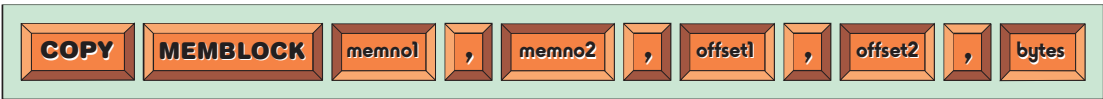
FIG-47.14

Data from one Memory Block can be Copied to Another



Copying is done using the COPY MEMBLOCK statement which has the format shown in FIG-47.15.

FIG-47.15 The COPY MEMBLOCK Statement



In the diagram:

memno1 is an integer value giving the ID of the source memory block.

memno2 is an integer value giving the ID of the destination memory block.

offset1 is an integer value giving the position within *memno1* at which copying is to start.

offset2 is an integer value giving the starting position within *memno2* at which copied data is to be placed.

bytes is an integer value giving the number of bytes to be copied.

For example, if we have previously created two memory blocks using the statements

```
MAKE MEMBLOCK 1, 20
MAKE MEMBLOCK 2, 50
```

and then placed data in memory block 1, the third 10 bytes of data in memory block 1 can be copied to the fourth group of 10 bytes in memory block 2 using the statement:

```
COPY MEMBLOCK 1, 2, 20, 30, 10
```

Strings and Memory Blocks

You may have noticed that memory blocks seem to be able to hold only numeric values; there would appear to be no way of writing a string. Strings present a slight difficulty since they can vary from containing no characters to containing thousands. If we are to store strings, then we either need to have some method of detecting the number of characters in a particular string, or we have to make all strings the same size. Although this second option might not seem very practical, it is, in fact, often used, particularly when many strings are being held. We'll start by finding out how to store a single string in a memory block. First, we allow the user to type in the string from the keyboard:

```
INPUT "Please enter your name ",name$
```

Now we need to discover how many characters are in the string so we can create a memory block of appropriate size:

```
characters = LEN(name$)
MAKE MEMBLOCK 1,characters
```

Since we can only transfer numbers to the memory block, we'll convert each character in the string to a numeric value and write it to the memory block:

```
FOR c = 1 TO characters
    value = ASC(MID$(name$,c))
    WRITE MEMBLOCK BYTE 1,c-1,value
NEXT c
```

To retrieve the information, we need to create an empty string, read each value from the block, convert it back to a character, and add the character to the end of our string we have just created:

```
result$=""
FOR c = 0 TO GET MEMBLOCK SIZE(1)-1
    ch$ = CHR$(MEMBLOCK BYTE(1,c))
    result$ = result$ + ch$
NEXT c
```

The program in LISTING-47.3 brings these statements together, storing and retrieving a string from a memory block.

LISTING-47.3

Storing String Values in
a Memory Block

```
REM *** Get string ***
INPUT "Enter name ",name$
REM *** Create block ***
characters = LEN(name$)
MAKE MEMBLOCK 1,characters
REM *** Store name in block ***
FOR c = 1 TO characters
    value = ASC(MID$(name$,c))
    WRITE MEMBLOCK BYTE 1,c-1,value
NEXT c
REM *** Read name back ***
result$=""
FOR c = 0 TO GET MEMBLOCK SIZE(1)-1
    ch$ = CHR$(MEMBLOCK BYTE(1,c))
    result$ = result$ + ch$
NEXT c
REM *** Display string read ***
PRINT "Value retrieved was ",result$
REM *** End program ***
WAIT KEY
END
```

Activity 47.3

Type in and test the program in LISTING-47.3 (*mem03.dbpro*).

In the next program we'll be a bit more ambitious. This time we'll use a memory block to store the top five scores in a game. Each entry will contain the player's name and his score.

This is where a fixed-size string will be useful. If we assume a name containing exactly 25 characters, then we know that every entry in our top-five list will be exactly 29 bytes in length; 25 for the name and 4 for the score. This, in turn, allows us to calculate the start position of every entry in the list; 0, 29, 56, etc.

But what about names that are less than 25 characters in length? We just pad them out with spaces to make them the required length. Names greater than 25 just won't be accepted.

So, the new program uses the following logic:

```
Set up memblock for 5 entries
FOR 5 times DO
  Get name
  Get score
  Add entry to memory block
ENDFOR
Display the memory block's contents
```

The logic is implemented in LISTING-47.4.

LISTING-47.4

Storing Top Score
Details in a Memory
Block

```
MAKE MEMBLOCK 1,29*5
FOR c = 1 TO 5
  name$ = GetName$()
  score = GetScore()
  AddDataToList(name$,score,c)
NEXT c
DisplayContentsOfMemBlock()
WAIT KEY
END

FUNCTION GetName$()
  INPUT "Enter name ",name$
  WHILE LEN(name$) > 25
    INPUT "Invalid name. Please re-enter ",name$
  ENDWHILE
  REM *** Pad name to 25 characters ***
  name$=name$+SPACE$(25 - LEN(name$))
ENDFUNCTION name$

FUNCTION GetScore()
  INPUT "Enter score ",score
  WHILE score < 0
    INPUT "Invalid score. Please re-enter ",score
  ENDWHILE
ENDFUNCTION score

FUNCTION AddDataToList(name$,score,post)
  byteposition = (post-1)*29
  REM *** Store name
  FOR c = 1 TO 25
```

continued on next page

LISTING-47.4
(continued)

Storing Top Score
Details in a Memory
Block

```
        value = ASC(MID$(name$,c))
        WRITE MEMBLOCK BYTE 1,byteposition+c-1,value
    NEXT c
    REM *** Store score ***
    WRITE MEMBLOCK DWORD 1,byteposition+25,score
ENDFUNCTION

FUNCTION DisplayContentsOfMemBlock()
    FOR c = 1 TO 5
        DisplayEntry(c)
    NEXT c
ENDFUNCTION

FUNCTION DisplayEntry(post)
    IF post < 1 OR post > 5
        EXITFUNCTION
    ENDIF
    byteposition = (post-1)*29
    REM *** Read name back ***
    name$=""
    FOR c = byteposition TO byteposition + 24
        ch$ = CHR$(MEMBLOCK BYTE(1,c))
        name$ = name$ + ch$
    NEXT c
    score = MEMBLOCK DWORD(1,byteposition + 25)
    PRINT name$, " ",score
ENDFUNCTION
```

Activity 47.4

Type in and test the program in LISTING-47.4 (*mem04.dbpro*).
Use the following data:

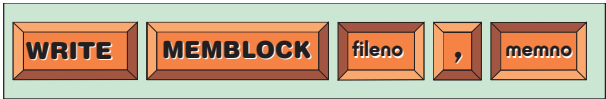
John Smith	3712
Liz Heron	2989
Avril Tait	2508
Jack Brown	1992
James Ladd	1894

The WRITE MEMBLOCK (to file) Statement

If we really did need to hold the top scores, we'd also need to save that information to a file so it would be available next time the game was played. To write the contents of a memory block to file we need to use the WRITE MEMBLOCK statement. The format for this statement is shown in FIG-47.16.

FIG-47.16

The WRITE
MEMBLOCK (to file)
Statement



In the diagram:

fileno

is an integer value giving the ID of the file
to which the memory block's data is to be written.

memno

is an integer value giving the ID of the memory
block whose contents are to be written to the file.

The statement writes not only the contents of the memory block, but also its size.

Before we can output to a file, it must be opened for writing (files were covered in Chapter 15):

```
OPEN TO WRITE 1, "toplist.dat"
```

The file will be created automatically in the current directory by the OPEN TO WRITE statement. The whole memory block can be written to the file with the single line:

```
WRITE MEMBLOCK 1,1
```

All that remains is to close the file:

```
CLOSE FILE 1
```

The lines have been placed in a function which also checks that the file does not exist before it is opened:

```
FUNCTION SaveListToFile()  
  IF FILE EXIST("toplist.dat")  
    DELETE FILE "toplist.dat"  
  ENDIF  
  OPEN TO WRITE 1, "toplist.dat"  
  WRITE MEMBLOCK 1,1  
  CLOSE FILE 1  
ENDFUNCTION
```

Activity 47.5

Add the function above to your last program and change the main section so that the function is called before the program terminates.

A file produced in this way is not limited to storing data from a memory block; additional information can be written to the file either before or after the data from the memory block. For example, it might prove useful to write to the file details such as the number of entries in the memory block, the size of an entry, the number of fields in one entry and a description of the fields within an entry. For our top-five list, that would involve saving the following information:

5	(entries in the block)
29	(the size of one entry)
2	(fields in one entry)
S25	(first field is a 25 character string)
D4	(second field is a double word which occupies 4 bytes)

The code required to write this information to the file is:

```
WRITE LONG 1,5  
WRITE LONG 1,29  
WRITE LONG 1,2  
WRITE STRING 1, "S25"  
WRITE STRING 1, "D4"
```

Activity 47.6

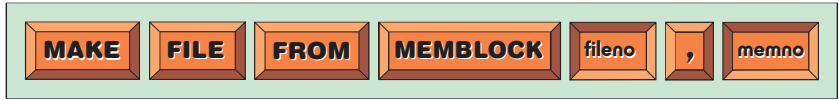
Modify the *SaveListToFile()* function so that the data described above is written to the file before the contents of the memory block. Also, add comments to the function.

The MAKE FILE FROM MEMBLOCK Statement

If the data to be held in the file is to be limited to only that coming from a single memory block, then there is an even simpler way to write that data to a file; the MAKE FILE FROM MEMBLOCK statement which has the format shown in FIG-47.17.

FIG-47.17

The MAKE FILE FROM MEMBLOCK Statement



In the diagram:

fileno

is an integer value giving the ID of the file which is to store the memory block's data.

memno

is an integer value giving the ID of the memory block whose data is to be copied.

We still need to open and close a file, but using this statement can be more convenient if nothing but the contents of the memory block are to be written to the file. Below is a second function for writing the top-five list to a file:

```
FUNCTION SaveListToFile2()  
  IF FILE EXIST("toplist2.mem")  
    DELETE FILE "toplist2.mem"  
  ENDIF  
  OPEN TO WRITE 1, "toplist2.mem"  
  MAKE FILE FROM MEMBLOCK 1,1  
  CLOSE FILE 1  
ENDFUNCTION
```

Activity 47.7

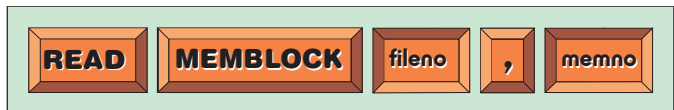
Add the new function above to your last program and modify the main section so that only this new function is used to save the memory block to a file.

The READ MEMBLOCK (from file) Statement

Of course, if we can write a memory block to a file, it makes sense that we should be able to read that data from the file and place it back in a memory block. To do this we need to use the READ MEMBLOCK statement which has the format shown in FIG-47.18.

FIG-47.18

The READ MEMBLOCK (from file) Statement



In the diagram:

fileno

is an integer value giving the ID of the file containing the data to be read.

memno

is an integer value giving the ID of the memory block which is to store the data read from the file. The memory block will be created automatically by this statement.

For the statement to operate correctly, we must ensure that the correct part of the file containing the data is accessed. For example, in *toplist.dat* the memory block is the sixth item within the file, so the other five must be read (or skipped) before any attempt is made to execute the READ MEMBLOCK statement.

If we create flexible code, we can read back almost any memory block by interpreting the information we find in the header.

We start by declaring global variables which will hold the header data:

```
GLOBAL records          \ no. of records in memblock
GLOBAL recsize          \ size of one record
GLOBAL fields           \ fields in one record
GLOBAL DIM fieldformat$(20) \ format of each field
GLOBAL DIM fieldsize(20)  \ size of each field
```

This allows up to 20 fields within a record.

The code below reads back all the information written to the file *toplist.dat*

```
FUNCTION ReadListFromFile()
  REM *** Open file for reading ***
  OPEN TO READ 1, "toplist.dat"
  REM *** Read header data ***
  READ LONG 1, records
  READ LONG 1, recsize
  READ LONG 1, fields
  FOR c = 1 TO fields
    READ STRING 1, field$
    fieldformat$(c) = LEFT$(field$,1)
    fieldsize(c) = VAL(RIGHT$(field$, LEN(field$)-1))
  NEXT c
  REM *** Read top 5 details ***
  READ MEMBLOCK 1,1
  REM *** Close the file ***
  CLOSE FILE 1
ENDFUNCTION
```

and an individual record within the memory block can be displayed using the following function:

```
FUNCTION DisplayEntry(post)
  IF post < 1 OR post > records
    EXITFUNCTION
  ENDIF
  byteposition = (post-1)*recsize
  REM *** Calculate position of record within memblock ***
  offset = (post-1)*recsize
  REM *** Display each field in the record ***
  FOR c = 1 TO fields
    SELECT fieldformat$(c)
      CASE "S"
        text$=""
        FOR k = byteposition TO byteposition +
          ↵ fieldsize(c)-1
          ch$ = CHR$(MEMBLOCK BYTE(1,offset))
          text$ = text$ + ch$
          INC offset
        NEXT k
        PRINT text$, " ";
      ENDCASE
      CASE "D"
        value = MEMBLOCK DWORD(1,offset)
        offset = offset + 4
        PRINT value, " ";
      ENDCASE
    ENDSELECT
  NEXT c
ENDFUNCTION
```

```

NEXT c
PRINT
ENDFUNCTION

```

Activity 47.8

Making use of the code above, create a complete program (*act4708.dbpro*) to read all the data held in *toplist.dat* and display the top five list.

The MAKE MEMBLOCK FROM FILE Statement

If a memory block is written to a file using the MAKE FILE FROM MEMBLOCK statement, then we can read that information back from the file, and recreate the memory block using the MAKE MEMBLOCK FROM FILE statement which has the format given in FIG-47.19.

FIG-47.19

The MAKE
MEMBLOCK FROM
FILE Statement



In the diagram:

fileno

is an integer value giving the ID of the file whose contents are to be loaded into a memory block.

memno

is an integer value giving the ID of the memory block which is to store the data retrieved from the file.

For example, we could read back the contents of *toplist2.mem* using the code:

```

OPEN TO READ 1, "toplist2.mem"
MAKE MEMBLOCK FROM FILE 1,1
CLOSE FILE 1

```

Activity 47.9

Create a program (*act4709.dbpro*) that reads the contents of *toplist2.mem* to a memory block and then displays the contents of that memory block.

Adding a New Top Score to our List

If a player achieves a new high score, then we'll want to add that score to the top five list (see FIG-47.20).

We can achieve the first step in this process - moving existing entries - using the COPY MEMBLOCK statement.

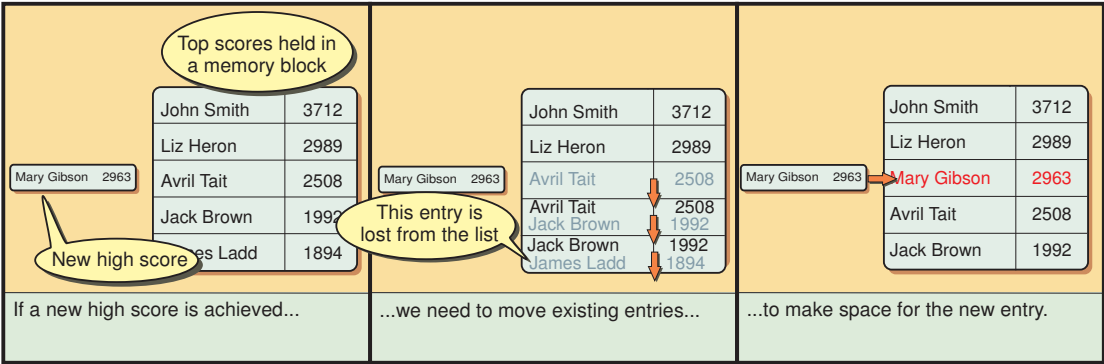
For example, to add *Mary Gibson* to our list, we could move entries 3 and 4 into positions 4 and 5 using the line:

```

COPY MEMBLOCK 1,1,2*29,3*29,2*29

```


FIG-47.20 Adding a New Entry into the Top Five List



Activity 47.10

Write a program which implements the following logic:

Read the contents of *toplist2.mem* into a memory block
Shift the contents of entries 3 and 4 into positions 4 and 5 in the list
Read details of a new name and score (Mary Gibson, 2963)
Place the new entry at position 3 in the top five list
Display the contents of the top five list.

Summary

- A pointer is a variable designed to contain an address.
- Use MAKE MEMBLOCK to dynamically allocate memory space.
- Use GET MEMBLOCK PTR to return the start address of a memory block.
- A pointer can be used to return values from a function.
- Use WRITE MEMBLOCK to write data to an area within a memory block.
- Use MEMBLOCK to retrieve information stored within a memory block.
- Use GET MEMBLOCK SIZE to discover the number of bytes in a memory block.
- Use DELETE MEMBLOCK to deallocate space assigned to a memory block.
- Use MEMBLOCK EXIST to check if a specified memory block exists.
- Use COPY MEMBLOCK to copy a part of one memory block to another memory block.
- Use WRITE MEMBLOCK to write the contents of a memory block to a file.
- The file specified in a WRITE MEMBLOCK statement may contain data other than that from the memory block.
- Use MAKE FILE FROM MEMBLOCK to create a file containing only the contents of a memory block.

- Use `READ MEMBLOCK` to read data from a file into a memory block.
- Use `MAKE MEMBLOCK FROM FILE` to read data from a file containing only data copied from a memory block into a new memory block.

Introduction

The most interesting use of memory blocks is undoubtedly employing them to store images, sounds, and 3D objects. Once the data representing such a component is stored in a memory block, we can manipulate that information in ways that are not possible using other techniques.

Bitmaps and Memory Blocks

Back in Chapter 13 we looked at bitmap objects. Bitmaps are storage areas for images. There are 32 different areas (numbered 0 to 31). Area 0 represents the screen - any image placed in area 0 will immediately be shown on the screen. To load an image into a bitmap area we use a statement such as

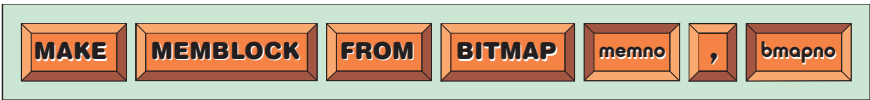
```
LOAD BITMAP "bbrush.bmp", 0
```

which loads an image file named *bbrush.bmp* onto the screen.

The MAKE MEMBLOCK FROM BITMAP Statement

To transfer the contents of a bitmap area to a memory block, we need to use the MAKE MEMBLOCK FROM BITMAP statement. The memory block specified should not already exist, since it will be created by the statement. The MAKE MEMBLOCK FROM BITMAP statement has the format shown in FIG-47.21.

FIG-47.21 The MAKE MEMBLOCK FROM BITMAP Statement



In the diagram:

- memno* is an integer value giving the ID to be assigned to the memory block created by this statement.
- bmapno* is an integer value giving the bitmap area that is to be copied to the memory block.

To copy the contents of the screen to memory block 1 we would use the statement:

```
MAKE MEMBLOCK FROM BITMAP 1, 0
```

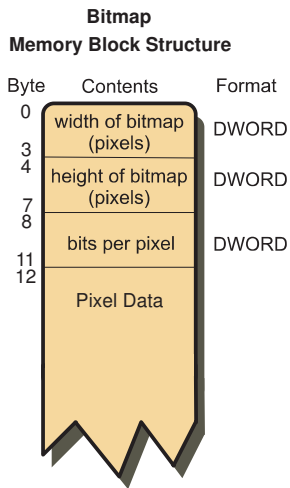
If we are to manipulate the contents of the memory block, we need to know the structure of the data held. A memory block containing a bitmap has the format shown in FIG-47.22.

As you can see, the first 12 bytes of the memory block contain details of the bitmap's width, height and bits per pixel. For most bitmap areas this will correspond to the details of the actual image that has been loaded into that area, but for bitmap area 0, which is the screen, the width, height and bits per pixel are determined by the values used in the SET DISPLAY MODE statement.

FIG-47.22

The Format of a Bitmap

Preliminary details such as those described for bitmaps is often generally referred to as **header** information.



The program in LISTING-47.5 demonstrates the transfer of the screen's image to a memory block and shows that the information read from the memory block matches that used in the SET DISPLAY MODE statement.

LISTING-47.5

Copying a Bitmap to a Memory Block

```
REM *** Set screen resolution ***
SET DISPLAY MODE 1280,1024,32

REM *** Copy screen data to a memory block ***
MAKE MEMBLOCK FROM BITMAP 1,0

REM *** Get details from memory block ***
width = MEMBLOCK DWORD(1,0)
height = MEMBLOCK DWORD(1,4)
bitsperpixel = MEMBLOCK DWORD(1,8)

REM *** Display details on the screen ***
PRINT "Screen is ",width," by ",height
PRINT "There are ",bitsperpixel," bits per pixel"

REM *** End program ***
WAIT KEY
END
```

Activity 47.11

Type in and test the program in LISTING-47.5 (*mem05.dbpro*).

Set the screen to 800 by 600 with 16 bits per pixel and check that the details returned match this figure.

Modify the program to display the number of bytes required by a single line of pixels on the screen.

Activity 47.12

Write a program (*act4712.dbpro*) which loads the image *bbrush.bmp* into bitmap area 1. By transferring the bitmap to a memory block, display the image's width, height and bits per pixel.

There are easier way of discovering these details! (see Chapter 13)

We can make use of the data in the memory block to actually change what appears on the screen by first changing values within the pixel data area of the memory

block and then writing the memory block back to bitmap area 0.

For example, if we assume a screen resolution of 800 by 600 and 16 bits per pixel, we could make the data representing the top-left pixel on the screen white by beginning with the line:

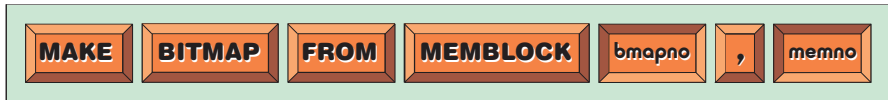
```
WRITE MEMBLOCK WORD 1,12,RGB(255,255,255)
```

The MAKE BITMAP FROM MEMBLOCK Statement

If the changes we have made to the memory block are to be reflected on the screen, we need to copy the memory block back to the bitmap area - area 0 in this case. This can be done using the MAKE BITMAP FROM MEMBLOCK statement whose format is shown in FIG-47.23.

FIG-47.23

The MAKE BITMAP
FROM MEMBLOCK
Statement



In the diagram:

bmapno

is an integer value giving the bitmap area into which the memory block is to be copied.

memno

is an integer value giving the ID of the memory block to be copied.

So, if we've changed the contents of memory block 1, we can write the updated version back to the screen using the line:

```
MAKE BITMAP FROM MEMBLOCK 0,1
```

The program in LISTING-47.6 starts by loading the black screen image into a memory block and changes the first pixel to white before rewriting the memory block's data to the screen.

LISTING-47.6

Manipulating a Bitmap's
Data

```
REM *** Set screen resolution ***
SET DISPLAY MODE 1280,1024,32
REM *** Copy screen to memory block ***
MAKE MEMBLOCK FROM BITMAP 1,0
PRINT "Screen captured"
WAIT KEY
REM *** Change first pixel in block to white ***
WRITE MEMBLOCK DWORD 1,12,RGB(255,255,255)
PRINT "Memblock changed"
WAIT KEY
REM *** Rewrite block to screen ***
MAKE BITMAP FROM MEMBLOCK 0,1
PRINT "Screen updated"
REM *** End program ***
WAIT KEY
END
```

Activity 47.13

Type in and test the program in LISTING-47.6 (*mem06.dbpro*).

Modify the program so that it is the 10th pixel on the first row that is made white.

Make sure that your monitor is adjusted to show all pixels that make up the screen output.

Mapping a Screen Position to a Memory Block Location

If we are to have absolute control over the manipulation of an image, we need to be able to calculate the position in the memory block of any pixel. Let's start by working out the position of any pixel on the first row.

Pixels are addressed using a column and row system, starting with column zero, row zero. So, pixel (0,0) has its data stored at position 12 in the memory block. If we are using a 32 bits / pixel setup, then pixel (1,0) starts at position 16, pixel (2,0) at position 20, etc. More generally, we can write that:

$\text{pixel}(x,0)$ starts at position $x*4 + 12$

If the image is 1280 pixels wide (as the screen is in the earlier examples) then the last pixel on the first line - pixel (1279,0) will start at position $1279*4 + 12 = 5128$ and therefore, the first pixel in the next row (0,1) must start at position 5132.

Activity 47.14

Modify your last program to make the last pixel of the first row and the first pixel of the second row white.

If each pixel requires 4 bytes of storage, and there are 1280 pixels in a row, then 1 row requires 5120 pixels. Now we can extend our formula to calculate the position of any pixel's data within the memory block to be:

$\text{pixel}(\text{col},\text{row}) = \text{col}*4 + \text{row}*5120 + 12$

Activity 47.15

Modify your last program to change the bottom right pixel on the screen - pixel (1279,1023) - to white

Our formula is still too specific - it assumes a resolution of 1280 by 1024 and 32 bits per pixel. A more general formula would be:

$\text{pixel}(\text{col},\text{row}) = \text{col} * \text{bytesperpixel} + \text{row} * \text{pixelsperrow} * \text{bytesperpixel} + 12$

The program in LISTING-47.7 loads an image, *bbrush.jpg*, onto the screen and then creates a negative version of the image by inverting the value of every pixel in the image.

LISTING-47.7

Calculating a Data Item's
Position in a Memory
Block

```
REM *** Set screen resolution ***
SET DISPLAY MODE 1280,1024,32
REM *** Load image ***
LOAD BITMAP "bbrush.jpg",0
REM *** Copy screen to memory block ***
MAKE MEMBLOCK FROM BITMAP 1,0
PRINT "Screen captured"
WAIT KEY

REM *** Modify every pixels' data ***
FOR row = 0 TO 1023
  FOR col = 0 TO 1279
    post = col*4 + row*5120 + 12
    currentvalue = MEMBLOCK DWORD(1,post)
```

continued on next page

LISTING-47.7

(continued)

Calculating a Data Item's
Position in a Memory
Block

```
        WRITE MEMBLOCK DWORD 1, post, NOT currentvalue
    NEXT col
NEXT row
PRINT "Memblock changed"
WAIT KEY

REM *** Copy new data to screen ***
MAKE BITMAP FROM MEMBLOCK 0,1
PRINT "Screen updated"
REM *** End program ***
WAIT KEY
END
```

Activity 47.16

Type in and test the program in LISTING-47.7 (*mem07.dbpro*).

Activity 47.17

Write a program (*act4717.dbpro*) which performs the following logic:

- Load the image *cactus.jpg* into bitmap area 1
- Copies the image to a memory block
- Exclusive ORs (XOR) the value 0X212121 to every pixel's data
- Copies the memory block's data to the screen.

Mapping the Mouse Position to a Memory Block Location

There are other ways to
achieve the same effect.

We can read the mouse pointer's position on the screen using MOUSEX and MOUSEY (see Chapter 27). By using the formula we developed earlier, we could use the mouse pointer to modify parts of the screen. The program in LISTING-47.8 changes the pixel at the mouse pointer's position to yellow using memory blocks.

LISTING-47.8

How the Mouse Position
Corresponds to the
Position of a Data Item in
a Memory Block

```
REM *** Set screen resolution ***
SET DISPLAY MODE 1280,1024,32

REM *** Copy screen to memory block ***
MAKE MEMBLOCK FROM BITMAP 1,0

REM *** Main loop ***
DO
    REM *** Get mouse coords ***
    col = MOUSEX()
    row = MOUSEY()
    REM *** Change matching area to yellow ***
    WRITE MEMBLOCK DWORD 1,row*5120 + col*4 + 12,RGB(255,255,0)
    REM *** Copy memory block back to screen ***
    MAKE BITMAP FROM MEMBLOCK 0,1
LOOP
REM *** End program ***
END
```

Activity 47.18

Type in and test the program in LISTING-47.8 (*mem08.dbpro*).

Modify the program so that a mouse button must be pressed before a pixel turns yellow.

Images and Memory Blocks

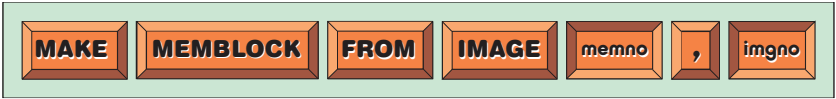
What we can do with bitmaps, we can also do to images; copying them to a memory block, changing the information held there, and then updating the image object with the modified data.

The MAKE MEMBLOCK FROM IMAGE Statement

To copy an image to a memory block we use the MAKE MEMBLOCK FROM IMAGE statement which has the format shown in FIG-47.24.

FIG-47.24

The MAKE
MEMBLOCK FROM
IMAGE Statement



In the diagram:

- memno* is an integer value giving the ID to be assigned to the memory block created by this statement.
- imgno* is an integer value giving the ID of the image to be copied to the memory block.

The memory block created by this statement has exactly the same format as that created with a bitmap, with the first 12 bytes giving the width, height and bits per pixel, before the image's own data is reached.

We could load an image into a memory block with statements such as:

```
LOAD IMAGE "bbrush.jpg", 1
MAKE MEMBLOCK FROM IMAGE 1, 1
```

The MAKE IMAGE FROM MEMBLOCK Statement

To reverse the process - copying a memory block to an image - we use the MAKE IMAGE FROM MEMBLOCK statement which has the format shown in FIG-47.25.

FIG-47.25

The MAKE IMAGE
FROM MEMBLOCK
Statement



In the diagram:

- imgno* is an integer value giving the ID of the image object to be created by this statement.
- memno* is an integer value giving the ID of the memory block to be copied.

The image ID specified in this statement should not be in use when the statement is executed.

In the program in LISTING-47.9 the bottlebrush image is used to texture a rotating cube. The image is then negated and reapplied to the cube.

LISTING-47.9

Manipulating an Image

```
REM *** Set screen resolution ***
SET DISPLAY MODE 1280,1024,32

REM *** Create textured object ***
LOAD IMAGE "bbrush.jpg",1
MAKE OBJECT CUBE 1,10
TEXTURE OBJECT 1,1

REM *** Copy image to memory block ***
MAKE MEMBLOCK FROM IMAGE 1,1
REM *** Negate image ***
width = MEMBLOCK DWORD(1,0)
height = MEMBLOCK DWORD(1,4)
bytes = MEMBLOCK DWORD(1,8)/8
FOR row = 0 TO height -1
  FOR col = 0 TO width-1
    post = col*bytes + row*bytes*width + 12
    currentvalue = MEMBLOCK DWORD(1,post)
    WRITE MEMBLOCK DWORD 1, post, NOT currentvalue
  NEXT col
NEXT row

REM *** main loop ***
DO
  REM *** Rotate object
  TURN OBJECT LEFT 1,1
  REM *** IF a key pressed, texture cube with new image ***
  IF INKEY$() <> ""
    DELETE IMAGE 1
    MAKE IMAGE FROM MEMBLOCK 1,1
    TEXTURE OBJECT 1,1
  ENDIF
  WAIT 5
LOOP
REM *** End program ***
END
```

Activity 47.19

Type in and test the program in LISTING-47.9 (*mem09.dbpro*).

Sounds and Memory Blocks

The next media object that can be stored and manipulated in a memory block is a sound object.

The MAKE MEMBLOCK FROM SOUND Statement

To transfer a loaded sound to a memory block, we need to use the MAKE MEMBLOCK FROM SOUND statement which has the format shown in FIG-47.26.

FIG-47.26

The MAKE MEMBLOCK FROM SOUND Statement



In the diagram:

memno

is an integer value giving the ID to be assigned to the memory block created by this statement.

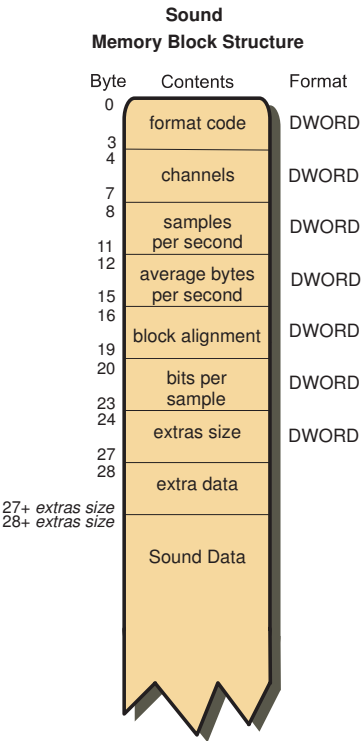
sndno

is an integer value giving the ID of the sound to be copied to the memory block.

The header information included at the start of a sound object is much more complex than that of a bitmap or image. Details such as the sound format being used, number of channels and sample rate are included. The actual structure of this header data is an agreed international standard and is shown in FIG-47.27.

FIG-47.27

The Data Structure Used for a Sound



Each field in the diagram is explained below:

- format code**

Many sound file formats use this same core data structure, but with different data compression algorithms. The actual type of compression used is identified here using a coded value registered with Microsoft.
- channels**

Identifies the number of sound channels used. For monaural the value here would be 1; for stereo, 2.
- samples per second**

Identifies how often the original sound has been sampled per second during the digitising session.
- average bytes per second**

Specifies the average data transfer rate in bytes. Normally, this will be
samples per second x block alignment
- block alignment**

This value gives the size of each data block sample. It should be equal to
channels x bits per sample

bits per sample	Specifies the number of bits used when sampling a single channel. Normally, 8 or 16.
extras size	Specifies the size in bytes of any information that has been added to this header section before the sound data is reached.
extra data	This is the area where any extra header information has been added. This area must be <i>extra size</i> bytes long.

The MAKE SOUND FROM MEMBLOCK Statement

Once sound data has been manipulated within a memory block it will need to be transferred back to a sound object, and this is achieved using the MAKE SOUND FROM MEMBLOCK statement which has the format shown in FIG-47.28.

FIG-47.28

The MAKE SOUND FROM MEMBLOCK Statement



In the diagram:

<i>sndno</i>	is an integer value giving the ID of the sound object to be created by this statement.
<i>memno</i>	is an integer value giving the ID of the memory block to be copied.

The program in LISTING-47.10 loads a sound file and displays the information held in its header area.

LISTING-47.10

Memory Blocks and Sound

```

REM *** Load and play sound ***
LOAD SOUND "welcome.wav",1
PLAY SOUND 1

REM *** Copy sound to memory block ***
MAKE MEMBLOCK FROM SOUND 1,1

REM *** Get data ***
channels = MEMBLOCK DWORD(1,4)
rate# = MEMBLOCK DWORD(1,8)/1000.0
avBperS = MEMBLOCK DWORD(1,12)
block = MEMBLOCK DWORD(1,16)
bits = MEMBLOCK DWORD(1,20)
extra = MEMBLOCK DWORD(1,24)

REM *** Display the information ***
PRINT "Channels used : ", channels
PRINT "Sample rate : ", rate#," kilohertz"
PRINT "Bytes per sec : ", avBperS
PRINT "Block size : ", block," bytes"
PRINT "Bits per sample: ", bits
PRINT "Extra data : ", extra," bytes"

REM *** End program ***
WAIT KEY
END

```

Activity 47.20

Type in and test the program in LISTING-47.10 (*mem10.dbpro*).

Modify the program so that within the memory block the sample rate is set to 44100 and the bytes per second to 88200. Delete sound 1 and recreate a sound 1 object from memory block 1. Play the sound. How have the changes altered the sound playback?

3D Objects and Memory Blocks

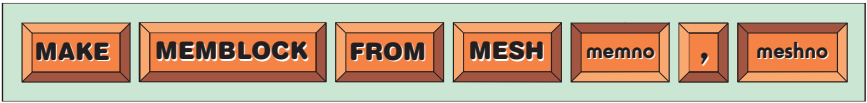
It is even possible to modify a mesh by transferring it to a memory block. However, the data structure used to represent mesh data is a complex one and we need to take care when manipulating its contents.

Mesh data is held in a format known as Flexible Vertex Flags or FVF which contains details of a vertex's coordinates, its normal, its diffuse value and UV texture offset.

The MAKE MEMBLOCK FROM MESH Statement

To convert a mesh to a memory block we need to use the MAKE MEMBLOCK FROM MESH statement which has the format shown in FIG-47.29.

FIG-47.29
The MAKE MEMBLOCK FROM MESH Statement

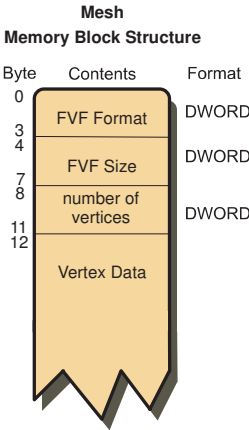


In the diagram:

- memno* is an integer value giving the ID to be assigned to the memory block created by this statement.
- meshno* is an integer value giving the ID of the mesh to be copied to the memory block.

The structure of the header information for a mesh is shown in FIG-47.30.

FIG-47.30
The Basic Header Structure for a Mesh



The fields in the header information are:

- FVF Format** There are several possible FVF formats. The integer value given here identifies the format used.

FVF Size	This field gives the size in bytes of each entry in the data area.
number of vertices	Specifies the number of vertices in the mesh. This corresponds to the number of entries in the Vertex Data area of the memory block.

The program in LISTING-47.11 creates a mesh from a cube, stores the data in a memory block, and displays the block's header details.

LISTING-47.11
Placing a Cube's Data in a Memory Block

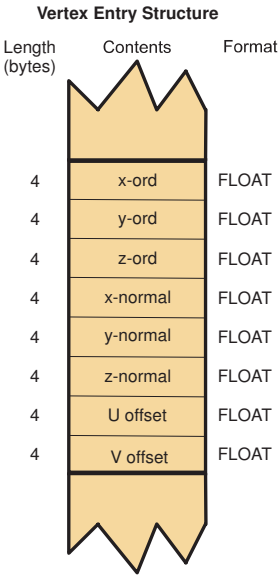
```
#CONSTANT cubeobj      1
#CONSTANT meshobj      1
#CONSTANT memobj       1
REM *** Manual screen updating ***
SYNC ON
REM *** Make cube ***
MAKE OBJECT CUBE cubeobj,10
SYNC
REM *** Convert cube to mesh ***
MAKE MESH FROM OBJECT meshobj,cubeobj
REM *** Copy mesh to memory block ***
MAKE MEMBLOCK FROM MESH memobj,meshobj
REM *** Get  and display details ***
FVFformat = MEMBLOCK DWORD (1,0)
entrysize = MEMBLOCK DWORD (1,4)
noofvertices = MEMBLOCK DWORD (1,8)
PRINT "FVF format      : ", FVFformat
PRINT "Entry size      : ", entrysize
PRINT "No. of vertices : ", noofvertices
SYNC
REM *** End program ***
WAIT KEY
END
```

Activity 47.21

Type in and test the program in LISTING-47.11 (*mem11.dbpro*).

Most DarkBasic Pro models seem to use format FVF 274 which holds, for each vertex, the information shown in FIG-47.31.

FIG-47.31
The Detailed Header Structure for a Mesh



The fields in the vertex block are:

x-ord, y-ord, z-ord

The coordinates of the vertex.

x-normal, y-normal, z-normal

The coordinates of the end point of the vertex's normal. These measurements are relative to the position of the vertex. A normal will usually be exactly 1 unit in length.

U offset, V offset

The UV offsets to be applied at that point when mapping a texture to the object.

The program in LISTING-47.12 displays the details of each vertex of the cube we created previously.

LISTING-47.12

Accessing Vertex
Details from a Memory
Block

```
#CONSTANT cubeobj      1
#CONSTANT meshobj      1
#CONSTANT memobj       1

REM *** Manual screen updating ***
SYNC ON

REM *** Create mesh from cube ***
MAKE OBJECT CUBE cubeobj,10
SYNC
MAKE MESH FROM OBJECT meshobj,cubeobj

REM *** Create memory block from mesh ***
MAKE MEMBLOCK FROM MESH memobj,meshobj

REM *** Get header details ***
FVFformat = MEMBLOCK DWORD(1,0)
entrysize = MEMBLOCK DWORD(1,4)
noofvertices = MEMBLOCK DWORD(1,8)

REM *** Get and display details of each vertex ***
FOR c = 0 TO noofvertices-1
    post = 12 + c*entrysize
    x# = MEMBLOCK FLOAT(1,post)
    y# = MEMBLOCK FLOAT (1,post+4)
    z# = MEMBLOCK FLOAT(1,post+8)
    nx# = MEMBLOCK FLOAT(1,post+12)
    ny# = MEMBLOCK FLOAT (1,post+16)
    nz# = MEMBLOCK FLOAT(1,post+20)
    U# = MEMBLOCK FLOAT(1,post+24)
    V# = MEMBLOCK FLOAT (1,post+28)
    PRINT "Coords (" ,STR$(x#,0) , " , " ,STR$(y#,0) , " , " ,STR$(z#,0) , " ) "
    PRINT "Normal (" ,STR$(nx#,0) , " , " ,STR$(ny#,0) , " , "
    PRINT " ,STR$(nz#,0) , " ) "
    PRINT "UV offsets U: " ,U# , " V: " ,V#
    IF c mod 10 = 0
        SYNC
        WAIT KEY
    ENDIF
NEXT c
SYNC

REM *** end program ***
WAIT KEY
END
```

Activity 47.22

Type in and test the program given in LISTING-47.12 (*mem12.dbpro*).

Another common format for 3D objects is FVF format 338. This structure is similar to 274 but includes a DWORD diffuse value in each vertex data block between the normal's z-ordinate and the U offset value.

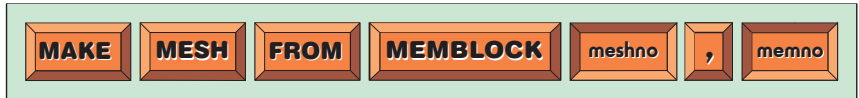
Each group of three vertices in the object are used to create a triangular polygon; the collection of polygons form the object.

The MAKE MESH FROM MEMBLOCK Statement

We can modify data in a memory block and then create a mesh from this new data. To create the new mesh from the memory block we use the MAKE MESH FROM MEMBLOCK statement which has the format shown in FIG-47.32.

FIG-47.32

The MAKE MESH
FROM MEMBLOCK
Statement



In the diagram:

meshno

is an integer value giving the ID of the mesh object to be created by this statement.

memno

is an integer value giving the ID of the memory block to be copied.

The program in LISTING-47.13 stores a cube's mesh in a memory block, modifies the coordinates of some of the vertices and then recreates a mesh and object from the modified data.

LISTING-47.13

Modifying a Mesh using
a Memory Block

```
#CONSTANT cubeobj      1
#CONSTANT rootobj      2
#CONSTANT meshobj      1
#CONSTANT memobj       1

SetUpScreen()

REM *** Create memory block from cube ***
MAKE OBJECT CUBE cubeobj,10
MAKE MESH FROM OBJECT meshobj,cubeobj
DELETE OBJECT cubeobj
MAKE MEMBLOCK FROM MESH memobj,meshobj
DELETE MESH meshobj

REM *** Get details of object ***
entrysize = MEMBLOCK DWORD(1,4)
noofvertices = MEMBLOCK DWORD(1,8)

REM *** Get coords of each vertex ***
FOR c = 0 TO noofvertices-1
    post = 12 + c*entrysize
    x# = MEMBLOCK FLOAT(1,post)
    y# = MEMBLOCK FLOAT (1,post+4)
    z# = MEMBLOCK FLOAT(1,post+8)
    REM *** Modify any vertices above y = 0 ***
    IF y# > 0
```

continued on next page

LISTING-47.13
(continued)

Modifying a Mesh using
a Memory Block

```
WRITE MEMBLOCK FLOAT 1,post,0
WRITE MEMBLOCK FLOAT 1,post+4,0
WRITE MEMBLOCK FLOAT 1,post+8,0
ENDIF
NEXT c

REM *** Make mesh from memory block ***
MAKE MESH FROM MEMBLOCK meshobj, memobj

REM *** Add mesh as limb to make it visible ***
MAKE OBJECT SPHERE rootobj,0
ADD LIMB rootobj,1,meshobj

REM *** Rotate object created ***
DO
    PITCH OBJECT UP 2,1
    WAIT 10
LOOP

REM *** End program ***
END

FUNCTION SetUpScreen()
    SET DISPLAY MODE 1280,1024,32
    AUTOCAM OFF
    POSITION CAMERA 0,0,-30
ENDFUNCTION
```

Activity 47.23

Type in and test the program in LISTING-47.13 (*mem13.dbpro*).

Modify the program so that, rather than change the coordinates of specific vertices, the program changes all vertex normal coordinates to (1,1,1).

How does this effect the cube?

The CHANGE MESH FROM MEMBLOCK Statement

Rather than delete the original mesh and then recreate it using MAKE MESH FROM MEMBLOCK, we can simply update the existing mesh using the CHANGE MESH FROM MEMBLOCK statement which has the format shown in FIG-47.33.

FIG-47.33
The CHANGE MESH FROM MEMBLOCK Statement



In the diagram:

meshno

is an integer value giving the ID of the mesh object to be updated by this statement.

memno

is an integer value giving the ID of the memory block to be copied.

Activity 47.24

Modify your last program so that the cube's mesh is updated rather than deleted and recreated.

Summary

- Bitmaps, images, sounds and meshes can be converted to or from memory blocks.
- Use `MAKE MEMBLOCK FROM BITMAP` to convert a bitmap's data to a memory block.
- A bitmap's data starts with header information detailing width, height and colour depth of the bitmap.
- Use `MAKE BITMAP FROM MEMBLOCK` to convert a memory block (containing the appropriate data) to a bitmap.
- Use `MAKE MEMBLOCK FROM IMAGE` to convert an image's data to a memory block.
- Use `MAKE IMAGE FROM MEMBLOCK` to convert memory block data to an image.
- Use `MAKE MEMBLOCK FROM SOUND` to convert a sound object to a memory block.
- A sound object's data starts with header information such as compression format, channels, and sample rate data.
- Use `MAKE SOUND FROM MEMBLOCK` to convert memory block data to a sound object.
- Use `MAKE MEMORY BLOCK FROM MESH` to convert a mesh's data to a memory block.
- A mesh's data starts with header information detailing the FVF format and the number of vertices in the mesh.
- Following the mesh header are details about each vertex within the mesh.
- Vertex data varies with the FVF format being used, but includes the vertex's spatial coordinates, normal coordinates and UV offset information.
- Use `MAKE MESH FROM MEMBLOCK` to convert memory block data to a mesh.
- Use `CHANGE MESH FROM MEMBLOCK` to update an existing mesh.

Solutions

Activity 47.1

No solution required.

Activity 47.2

No solution required.

Activity 47.3

No solution required.

Activity 47.4

No solution required.

Activity 47.5

The main section should be changed to:

```
MAKE MEMBLOCK 1,29*5
FOR c = 1 TO 5
    name$ = GetName$()
    score = GetScore()
    AddDataToList (name$, score, c)
NEXT c
DisplayContentsOfMemBlock()
SaveListToFile()
WAIT KEY
END
```

Activity 47.6

The code for function *SaveListToFile()* should be changed to:

```
FUNCTION SaveListToFile()
    REM *** IF file exists, destroy it ***
    IF FILE EXIST("TopList.dat")
        DELETE FILE "Toplist.dat"
    ENDIF
    REM *** Create file for writing ***
    OPEN TO WRITE 1, "Toplist.dat"
    REM *** Write header info to file ***
    WRITE LONG 1,5
    WRITE LONG 1,29
    WRITE LONG 1,2
    WRITE STRING 1, "S25"
    WRITE STRING 1, "D4"
    REM *** Write memblock to file ***
    WRITE MEMBLOCK 1,1
    REM *** Close file ***
    CLOSE FILE 1
ENDFUNCTION
```

Activity 47.7

The updated version of the main section is:

```
MAKE MEMBLOCK 1,29*5
FOR c = 1 TO 5
    name$ = GetName$()
    score = GetScore()
    AddDataToList (name$, score, c)
NEXT c
DisplayContentsOfMemBlock()
SaveListToFile2()
```

```
WAIT KEY
END
```

Activity 47.8

```
GLOBAL size
GLOBAL records
GLOBAL recsize
GLOBAL fields
GLOBAL DIM fieldformat$(20)
GLOBAL DIM fieldsize(20)
```

```
ReadListFromFile()
DisplayContentsOfMemBlock()
WAIT KEY
END
```

```
FUNCTION ReadListFromFile()
    REM *** Open file for reading ***
    OPEN TO READ 1, "toplist.dat"
    REM *** Read header data ***
    READ LONG 1,records
    READ LONG 1,recsize
    READ LONG 1,fields
    FOR c = 1 TO fields
        READ STRING 1,field$
        fieldformat$(c) = LEFT$(field$,1)
        fieldsize(c) = VAL(RIGHT$(field$,
            LEN(field$)-1))
    NEXT c
    REM *** Read top 5 details ***
    READ MEMBLOCK 1,1
    REM *** Close the file ***
    CLOSE FILE 1
ENDFUNCTION
```

```
FUNCTION DisplayContentsOfMemBlock()
    FOR c = 1 TO records
        DisplayEntry(c)
    NEXT c
ENDFUNCTION
```

```
FUNCTION DisplayEntry(post)
    IF post < 1 OR post > records
        EXITFUNCTION
    ENDIF
    byteposition = (post-1)*recsize
    REM *** Calculate position of entry ***
    offset = (post-1)*recsize
    FOR c = 1 TO fields
        SELECT fieldformat$(c)
            CASE "S"
                text$=""
                FOR k = byteposition TO
                    byteposition + fieldsize(c)-1
                    ch$ = CHR$(MEMBLOCK
                        LEN(BYTE(1,offset))
                        text$ = text$ + ch$
                    INC offset
                NEXT c
                PRINT text$," ";
            ENDCASE
            CASE "D"
                value = MEMBLOCK DWORD(1,offset)
                offset = offset + 4
                PRINT value," ";
            ENDCASE
        ENDSELECT
    NEXT c
    PRINT
ENDFUNCTION
```

Activity 47.9

```
OPEN TO READ 1, "toplist2.mem"
MAKE MEMBLOCK FROM FILE 1,1
CLOSE FILE 1
DisplayContentsOfMemBlock()
WAIT KEY
END

FUNCTION DisplayContentsOfMemBlock()
  FOR c = 1 TO 5
    DisplayEntry(c)
  NEXT c
ENDFUNCTION

FUNCTION DisplayEntry(post)
  IF post < 1 OR post > 5
    EXITFUNCTION
  ENDIF
  byteposition = (post-1)*29
  REM *** Read name back ***
  name$=""
  FOR c = byteposition TO byteposition + 24
    ch$ = CHR$(MEMBLOCK BYTE(1,c))
    name$ = name$ + ch$
  NEXT c
  score = MEMBLOCK DWORD(1,byteposition +
    ⤵25)
  PRINT name$, " ",score
ENDFUNCTION
```

The code is much simpler than that in Activity 47.8, but is designed to handle a specific memblock structure only.

Activity 47.10

```
OPEN TO READ 1, "TopList2.mem"
MAKE MEMBLOCK FROM FILE 1,1
CLOSE FILE 1
REM *** Move entries 3 and 4 ***
COPY MEMBLOCK 1,1,2*29,3*29, 2*29
name$ = GetName$()
score = GetScore()
AddDataToList(name$,score,3)
DisplayContentsOfMemBlock()
WAIT KEY
END

FUNCTION GetName$()
  INPUT "Enter name",name$
  WHILE LEN(name$) > 25
    INPUT "Invalid name. Please
    ⤵re-enter",name$
  ENDWHILE
  REM *** Pad name to 25 characters ***
  name$=name$+SPACE$(25 - LEN(name$))
ENDFUNCTION name$

FUNCTION GetScore()
  INPUT "Enter score ",score
  WHILE score < 0
    INPUT "Invalid score. Please
    ⤵re-enter",score
  ENDWHILE
ENDFUNCTION score

FUNCTION AddDataToList(name$,score,post)
  byteposition = (post-1)*29
  REM *** Store name
  FOR c = 1 TO 25
    value = ASC(MID$(name$,c))
    WRITE MEMBLOCK BYTE 1,
    ⤵byteposition+c-1,value
  NEXT c
  REM *** Store score ***
  WRITE MEMBLOCK DWORD 1,
```

```
⤵byteposition+25,score
ENDFUNCTION

FUNCTION DisplayContentsOfMemBlock()
  FOR c = 1 TO 5
    DisplayEntry(c)
  NEXT c
ENDFUNCTION

FUNCTION DisplayEntry(post)
  IF post < 1 OR post > 5
    EXITFUNCTION
  ENDIF
  byteposition = (post-1)*29
  REM *** Read name back ***
  name$=""
  FOR c = byteposition TO byteposition+24
    ch$ = CHR$(MEMBLOCK BYTE(1,c))
    name$ = name$ + ch$
  NEXT c
  score = MEMBLOCK DWORD(1,byteposition+25)
  PRINT name$, " ",score
ENDFUNCTION
```

Activity 47.11

To use the display mode requested, the line

```
SET DISPLAY MODE 1280,1024,32
```

should be changed to:

```
SET DISPLAY MODE 800,600,16
```

The modified version of the program is:

```
REM *** Set screen resolution ***
SET DISPLAY MODE 800,600,16
REM *** Copy screen data to memory block ***
MAKE MEMBLOCK FROM BITMAP 1,0
REM *** Get details from memory block ***
width = MEMBLOCK DWORD(1,0)
height = MEMBLOCK DWORD(1,4)
bitsperpixel = MEMBLOCK DWORD(1,8)
REM *** Display details on the screen ***
PRINT "Screen is ",width," by ",height
PRINT "There are ",bitsperpixel,
⤵" bits per pixel"
PRINT "A line of pixels requires ",width *
⤵bitsperpixel / 8, " bytes"
REM *** End program ***
WAIT KEY
END
```

Activity 47.12

```
REM *** Set screen resolution ***
SET DISPLAY MODE 1280,1024,32
REM *** Load bitmap ***
LOAD BITMAP "bbrush.jpg",1
REM *** Copy screen data to memory block ***
MAKE MEMBLOCK FROM BITMAP 1,1
REM *** restore screen as default bitmap ***
SET CURRENT BITMAP 0
REM *** Get details from memory block ***
width = MEMBLOCK DWORD(1,0)
height = MEMBLOCK DWORD(1,4)
bitsperpixel = MEMBLOCK DWORD(1,8)
REM *** Display details on the jpeg ***
PRINT "Picture is ",width," by ",height
PRINT "There are ",bitsperpixel,
⤵" bits per pixel"
REM *** End program ***
WAIT KEY
END
```

Activity 47.13

Only the offset value must be changed (and the comment):

```
REM *** Set screen resolution ***
SET DISPLAY MODE 1280,1024,32
REM *** Copy screen to memory block ***
MAKE MEMBLOCK FROM BITMAP 1,0
PRINT "Screen captured"
WAIT KEY
REM *** Change pixel 10 to white ***
WRITE MEMBLOCK DWORD 1,48,RGB(255,255,255)
PRINT "Memblock changed"
WAIT KEY
REM *** Rewrite block to screen ***
MAKE BITMAP FROM MEMBLOCK 0,1
PRINT "Screen updated"
REM *** End program ***
WAIT KEY
END
```

Activity 47.14

```
REM *** Set screen resolution ***
SET DISPLAY MODE 1280,1024,32
REM *** Copy screen to memory block ***
MAKE MEMBLOCK FROM BITMAP 1,0
PRINT "Screen captured"
WAIT KEY
REM *** (1279,0) and (0,1) to white ***
WRITE MEMBLOCK DWORD 1,5128,RGB(255,255,255)
WRITE MEMBLOCK DWORD 1,5132,RGB(255,255,255)
PRINT "Memblock changed"
WAIT KEY
REM *** Rewrite block to screen ***
MAKE BITMAP FROM MEMBLOCK 0,1
PRINT "Screen updated"
REM *** End program ***
WAIT KEY
END
```

Activity 47.15

```
REM *** Set screen resolution ***
SET DISPLAY MODE 1280,1024,32
REM *** Copy screen to memory block ***
MAKE MEMBLOCK FROM BITMAP 1,0
PRINT "Screen captured"
WAIT KEY
REM *** Change (1279,1023) to white ***
WRITE MEMBLOCK DWORD 1,1279*4 + 1023*5120,
↳ RGB(255,255,255)
PRINT "Memblock changed"
WAIT KEY
REM *** Rewrite block to screen ***
MAKE BITMAP FROM MEMBLOCK 0,1
PRINT "Screen updated"
REM *** End program ***
WAIT KEY
END
```

Activity 47.16

No solution required.

Activity 47.17

```
REM *** Set screen resolution ***
SET DISPLAY MODE 1280,1024,32
REM *** Load image ***
LOAD BITMAP "cactus.jpg",1
REM *** Return output to screen ***
SET CURRENT BITMAP 0
REM *** Copy bitmap to memory block ***
```

```
MAKE MEMBLOCK FROM BITMAP 1,1
REM *** Get bitmap details ***
width = MEMBLOCK DWORD(1,0)
height = MEMBLOCK DWORD(1,4)
bitsperpixel = MEMBLOCK DWORD(1,8)
REM *** Modify every pixel's data ***
FOR row = 0 TO height-1
  FOR col = 0 TO width-1
    post = col*4 + row*4*width + 12
    value = MEMBLOCK DWORD(1,post)
    ↳ XOR 0X212121
    WRITE MEMBLOCK DWORD 1,post,value
  NEXT col
NEXT row
PRINT "Memblock changed"
WAIT KEY
REM *** Copy new data to screen ***
MAKE BITMAP FROM MEMBLOCK 0,1
PRINT "Screen updated"
REM *** End program ***
WAIT KEY
END
```

Activity 47.18

```
REM *** Set screen resolution ***
SET DISPLAY MODE 1280,1024,32
REM *** Copy screen to memory block ***
MAKE MEMBLOCK FROM BITMAP 1,0
REM *** Main loop ***
DO
  REM *** Get mouse coords ***
  col = MOUSEX()
  row = MOUSEY()
  REM *** On mousedown, show yellow ***
  IF MOUSECLICK() <> 0
    WRITE MEMBLOCK DWORD 1,row*5120 +
    ↳ col*4 + 12,RGB(255,255,0)
    REM *** Copy memblock to screen ***
    MAKE BITMAP FROM MEMBLOCK 0,1
  ENDIF
LOOP
REM *** End program ***
END
```

Activity 47.19

No solution required.

Activity 47.20

```
REM *** Load and play sound ***
LOAD SOUND "welcome.wav",1
PLAY SOUND 1
REM *** Get data ***
MAKE MEMBLOCK FROM SOUND 1,1
channels = MEMBLOCK DWORD(1,4)
rate# = MEMBLOCK DWORD(1,8)/1000.0
avBperS = MEMBLOCK DWORD(1,12)
block = MEMBLOCK DWORD(1,16)
bits = MEMBLOCK DWORD(1,20)
extra = MEMBLOCK DWORD(1,24)
REM *** Display the information ***
PRINT "Channels used : ", channels
PRINT "Sample rate : ", rate#,"
kilohertz"
PRINT "Bytes per sec : ", avBperS
PRINT "Block size : ", block," bytes"
PRINT "Bits per sample: ", bits
PRINT "Extra data : ", extra," bytes"
REM *** Modify sound ***

WRITE MEMBLOCK DWORD 1,8,44100
WRITE MEMBLOCK DWORD 1,12,88200
DELETE SOUND 1
```

```
REM *** Reload modified sound ***
MAKE SOUND FROM MEMBLOCK 1,1
WAIT KEY
PLAY SOUND 1
```

```
REM *** End program ***
WAIT KEY
END
```

Playback sounds speeded up.

should be removed from the program and the line

```
MAKE MESH FROM MEMBLOCK meshobj, memobj
```

changed to

```
CHANGE MESH FROM MEMBLOCK meshobj, memobj
```

Activity 47.21

No solution required.

Activity 47.22

No solution required.

Activity 47.23

```
#CONSTANT cubeobj      1
#CONSTANT rootobj      2
#CONSTANT meshobj      1
#CONSTANT memobj       1
SetUpScreen()
REM *** Create memory block from cube ***
MAKE OBJECT CUBE cubeobj,10
MAKE MESH FROM OBJECT meshobj,cubeobj
DELETE OBJECT cubeobj
MAKE MEMBLOCK FROM MESH memobj,meshobj
DELETE MESH meshobj
REM *** Get details of object ***
entrysize = MEMBLOCK DWORD(1,4)
noofvertices = MEMBLOCK DWORD(1,8)
REM *** Get coords of each vertex ***
FOR c = 0 TO noofvertices-1
    REM *** Calc start of normal data ***
    post = 12 + c*entrysize + 12
    REM *** Modify all normals to (1,1,1) ***
    WRITE MEMBLOCK FLOAT 1,post,1
    WRITE MEMBLOCK FLOAT 1,post+4,1
    WRITE MEMBLOCK FLOAT 1,post+8,1
NEXT c
REM *** Make mesh from memory block ***
MAKE MESH FROM MEMBLOCK meshobj, memobj
REM *** Add mesh as limb to make it visible
***
MAKE OBJECT SPHERE rootobj,0
ADD LIMB rootobj,1,meshobj
REM *** Rotate object created ***
DO
    PITCH OBJECT UP 2,1
    WAIT 10
LOOP
REM *** End program ***
END

FUNCTION SetUpScreen()
    SET DISPLAY MODE 1280,1024,32
    AUTOCAM OFF
    POSITION CAMERA 0,0,-30
ENDFUNCTION
```

The cube changes shade as it rotates.

Activity 47.24

The line

```
DELETE MESH meshobj
```


Open Dynamics Engine

Adding Physical Laws to a 3D Object

Applying Friction

Applying Gravity

Dynamic and Static ODE Volumes

Linear and Angular Velocities

Surface and Contact Properties

Introduction

In virtual space objects stay still until they are moved. Things don't fall, bounce, skid or get damaged. There's no gravity and no friction. Objects don't have mass; a large object won't push a smaller one out of the way as they collide. In fact, unless we implement collision checks, objects can pass straight through each other without any interaction.

Although this lack of real-world laws can give us great freedom in what happens in our 3D world, it also makes simulations of real life difficult.

So to help things along, DarkBASIC Pro has some basic commands built-in which give us a hint at what is possible when the laws of physics are applied to our objects.

The ODE statements in DarkBASIC Pro are undocumented and although they are still present in DarkBASIC Pro version 1.062, they need to be loaded separately.

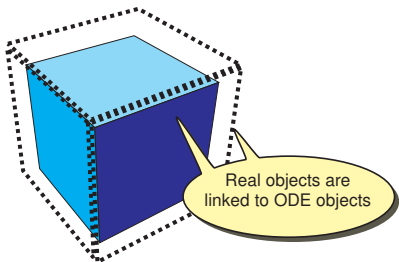
The details given in this chapter are based on the results obtained from various DarkBASIC Pro (v1.059) test programs and from other ODE documentation. If you're impressed by these features, get hold of The Game Creator's DarkPhysics add-on for really professional results.

Basic ODE Statements

Every visual 3D object that we want to react in a realistic fashion must be linked to its own ODE object (see FIG-48.1).

FIG-48.1

Invisible ODE objects
are linked to Normal
3D Objects



While our own 3D objects are visible, ODE objects are always invisible. But when an ODE object reacts to physical laws, it causes the visible object to which it is attached to react in the same way.

The ODE CREATE DYNAMIC BOX Statement

The physics box to be linked to a normal 3D object is created using the ODE CREATE DYNAMIC BOX statement which has the format shown in FIG-48.2.

FIG-48.2

The ODE CREATE DYNAMIC BOX Statement



In the diagram:

objno

is an integer value giving the ID of the 3D object to which the ODE box is to be linked. Only one ODE box can be linked to each 3D object.

width

is a real number giving the width of the ODE box.

height

is a real number giving the height of the ODE box.

depth

is a real number giving the depth of the ODE box.

For example, if we create a normal 3D cube using the line

```
CREATE OBJECT CUBE 1, 10
```

we could link an ODE box to that cube using the line

```
ODE CREATE DYNAMIC BOX 1
```

The ODE box would be exactly the same size as the cube which it surrounds.

Should we want to make the ODE box twice the size of the cube, then we could use the line

```
ODE CREATE DYNAMIC BOX 1, 20,20,20
```

making the ODE box 20 units in all three dimensions.

The ODE START Statement

The ODE physics engine does a massive amount of calculations to ensure that all ODE boxes (and other shapes - see later) react in a realistic way. The engine needs to be initialised and this is done using the ODE START statement which has the format shown in FIG-48.3.

FIG-48.3

The ODE START
Statement



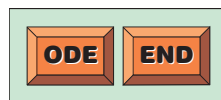
This statement must appear in your program before any other ODE related statement.

The ODE END Statement

When we no longer require the physics engine to operate, the ODE END statement should be executed. This statement has the form shown in FIG-48.4.

FIG-48.4

The ODE START
Statement



The ODE UPDATE Statement

The physics engine needs to update the screen in order to reposition the 3D models

as a result of gravitational pull or other forces acting on the objects. This is achieved by using the ODE UPDATE statement which has the format shown in FIG-48.5.

FIG-48.5

The ODE UPDATE
Statement



Now we're ready to make a start on using ODE. In LISTING-48.1 a cube falls from above, under the influence of gravity.

LISTING-48.1

Getting Started with ODE

```
ScreenSetUp()  
REM *** Make cube ***  
MAKE OBJECT CUBE 1,10  
POSITION OBJECT 1,0,25,0  
WAIT KEY  
REM *** Start Physics engine ***  
ODE START  
REM *** Link ODE box to cube ***  
ODE CREATE DYNAMIC BOX 1  
REM *** Update screen ***  
DO  
    ODE UPDATE  
LOOP  
REM *** End program ***  
ODE END  
END  
  
FUNCTION ScreenSetUp()  
    SET DISPLAY MODE 1280,1024,32  
    COLOR BACKDROP 0  
    BACKDROP ON  
    AUTOCAM OFF  
    POSITION CAMERA 0,10,-50  
    POINT CAMERA 0,0,0  
ENDFUNCTION
```

Activity 48.1

Type in and test the program given in LISTING-48.1 (*ODE01.dbpro*).

The ODE SET WORLD GRAVITY Statement

The ODE world comes with gravity built in - as we saw in the last program. But the force of gravity can be changed (and even made to act in other directions) using the SET WORLD GRAVITY statement which has the format shown in FIG-48.6.

FIG-48.6

The SET WORLD GRAVITY Statement



In the diagram:

xforce

is a real number giving the gravitational force applied along the x-axis.

yforce

is a real number giving the gravitational force applied along the y-axis.

zforce

is a real number giving the gravitational force applied along the z-axis.

The default gravitational force is down the y-axis only and is equivalent to the statement:

```
ODE SET WORLD GRAVITY 0,-9.81,0
```

Activity 48.2

In your last program, add the following line immediately after ODE START

```
ODE SET WORLD GRAVITY 0,-2,0
```

Check out the following other settings:

<i>xforce</i>	<i>yforce</i>	<i>zforce</i>
-2	0	0
0	0	2
0	-2	2
-2	-2	2

The ODE CREATE STATIC BOX Statement

If we create a plane to represent the ground, perhaps we can get the falling box to stop when it hits the ground's surface. We'll test this by adding the lines

```
MAKE OBJECT PLAIN 2, 100,100  
XROTATE OBJECT 2, 90
```

to the previous program.

Activity 48.3

Add the above lines near the start of your last program.

Return the gravity to a setting of 0,-2,0.

Add a WAIT KEY statement before starting the ODE engine. Does the cube stop when it hits the ground?

The cube goes straight through the ground plane. Of course, this is because the plane object does not have an ODE box.

Activity 48.4

Add the line

```
ODE CREATE DYNAMIC BOX 2
```

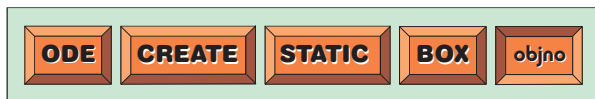
at the appropriate point in your program. Does the cube stop when it hits the ground?

And so another problem becomes apparent! Putting a dynamic box around any object means it is affected by gravity just like the cube.

When we want an object to remain in a fixed position and not be affected by normal forces and yet still be part of the ODE universe (so that other objects will react to coming into contact with it), we need to build a static ODE box around the fixed object rather than a dynamic box. This is done using the ODE CREATE STATIC BOX statement which has the format shown in FIG-48.7.

FIG-48.7

The ODE CREATE
STATIC BOX Statement



In the diagram:

objno

is an integer value giving the ID of the 3D object to which the ODE box is to be linked.

For example, we could link a static collision box to object 2 using the line:

```
ODE CREATE STATIC BOX 2
```

Activity 48.5

Modify the last program to link a static box to the plane object.

Does the ground and cube react correctly now?

Modify the dynamic box around the cube so that it has width, height and depth of 20 units.

What happens when the cube falls?

The ODE CREATE DYNAMIC SPHERE Statement

To make a collision appear accurate, the shape of the ODE volume should match as closely as possible that of the 3D object to which it is attached. For example, linking a dynamic box to a 3D sphere will not give accurate collision results under some circumstances. To handle this there are several other dynamic volumes which can be linked to a 3D object. A dynamic sphere can be linked using the ODE CREATE DYNAMIC SPHERE statement. This statement has the format shown in FIG-48.8.

FIG-48.8

The ODE MAKE
DYNAMIC SPHERE
Statement



In the diagram:

objno

is an integer value giving the ID of the 3D object to which the ODE sphere is to be linked.

Activity 48.6

Change the ODE volume around the 3D cube to a sphere. How does this affect the collision?

The ODE CREATE DYNAMIC CYLINDER Statement

Another option when creating a dynamic volume is the cylinder. This is created

using the ODE CREATE DYNAMIC CYLINDER statement which has the format shown in FIG-48.9.

FIG-48.9

The ODE CREATE
DYNAMIC CYLINDER
Statement



In the diagram:

objno

is an integer value giving the ID of the 3D object to which the ODE cylinder is to be linked.

The program in LISTING-48.2 creates a cube, a sphere and a cylinder, then links each object to the appropriate ODE dynamic volume before letting all three drop to the ground.

LISTING-48.2

Dropping Multiple
Objects

```
ScreenSetUp()

REM *** Make cube ***
MAKE OBJECT CUBE 1,10
POSITION OBJECT 1, 0,25,0
REM *** Make sphere ***
MAKE OBJECT SPHERE 2,5,20,20
POSITION OBJECT 2,-10,25,0
REM *** Make cylinder ***
MAKE OBJECT CYLINDER 3,8
POSITION OBJECT 3, 10,25,0

REM *** Make and texture ground ***
MAKE OBJECT PLAIN 4,100,100
LOAD IMAGE "grid8by8.bmp",1
TEXTURE OBJECT 4,1
XROTATE OBJECT 4,-90
REM *** Wait before starting ODE ***
WAIT KEY

REM *** Start ODE ***
ODE START
ODE SET WORLD GRAVITY 0,-2,0

REM Add ODE volumes ***
ODE CREATE DYNAMIC BOX 1
ODE CREATE DYNAMIC SPHERE 2
ODE CREATE DYNAMIC CYLINDER 3
ODE CREATE STATIC BOX 4

REM *** Continually update physics ***
DO
    ODE UPDATE
LOOP
REM *** End program ***
ODE END
END

FUNCTION ScreenSetUp()
    SET DISPLAY MODE 1280,1024,32
    COLOR BACKDROP 0
    BACKDROP ON
    AUTOCAM OFF
    POSITION CAMERA 0,10,-50
    POINT CAMERA 0,0,0
ENDFUNCTION
```

Activity 48.7

Type in and test the program given in LISTING-48.2 (*ODE02.dbpro*).

The ODE CREATE DYNAMIC TRIANGLE MESH Statement

If a complex model has to be linked to an ODE volume, this can be done with the standard shapes already mentioned, but alternatively a more accurate volume can be produced using the ODE CREATE DYNAMIC TRIANGLE MESH statement which has the format shown in FIG-48.10.

FIG-48.10

The ODE CREATE
DYNAMIC TRIANGLE
MESH Statement



In the diagram:

objno

is an integer value giving the ID of the 3D object to which the ODE mesh is to be linked.

In LISTING-48.3 the Hivebrain model uses an ODE mesh to detect its collision with the ground.

LISTING-48.3

Using a Triangle Mesh

```
ScreenSetUp()
REM *** Load model ***
LOAD OBJECT "H-Alien Hivebrain-Move.x",1
SCALE OBJECT 1, 500,500,500
POSITION OBJECT 1, 0,30,0

REM *** Make ground ***
MAKE OBJECT PLAIN 4,100,100
LOAD IMAGE "grid8by8.bmp",1
TEXTURE OBJECT 4,1
XROTATE OBJECT 4,-90

REM *** Wait before starting ODE ***
WAIT KEY

REM *** Start ODE ***
ODE START
ODE SET WORLD GRAVITY 0,-4,0

REM *** Use triangle mesh on model ***
ODE CREATE DYNAMIC TRIANGLE MESH 1

REM *** Use box on ground ***
ODE CREATE STATIC BOX 4
DO
    ODE UPDATE
LOOP
ODE END
END

FUNCTION ScreenSetUp()
    SET DISPLAY MODE 1280,1024,32
    COLOR BACKDROP 0
    BACKDROP ON
    AUTOCAM OFF
    POSITION CAMERA 0,10,-50
    POINT CAMERA 0,0,0
ENDFUNCTION
```

Activity 48.8

Type in and test the program in LISTING-48.3 (*ODE03.dbpro*).

How accurate is the collision detection?

Change the ODE volume used on the model to a box. Is this more accurate?

The ODE SET WORLD STEP

Every time the physics engine recalculates the position of the ODE volumes, it assumes a specific amount of time has passed since the last calculation. To give an accurate simulation, this time gap must be small - just a fraction of a second. To set this time gap, we can use the ODE SET WORLD STEP statement which has the format shown in FIG-48.11.

FIG-48.11

The ODE SET WORLD
STEP Statement



In the diagram:

time

is a real number representing the time interval between each calculation. Values between 0.04 and 1.7 work.

The effect of this statement is to speed up or slow down movement (since the time scale is changed) as well as affecting the accuracy of the movements.

Activity 48.9

In your last program add the line

```
ODE SET WORLD STEP 0.05
```

immediately after the ODE START statement.

Change the step size to 0.2.

How does this change the overall effect?

The ODE CREATE STATIC TRIANGLE MESH Statement

The program in LISTING-48.4 has two static objects; the ground and a cone placed on the ground. A cube then falls onto the cone.

LISTING-48.4

Adding a Realistic ODE
Volume for the Cone

```
ScreenSetUp ()
```

```
REM *** Make objects ***  
MAKE OBJECT CUBE 1,7  
POSITION OBJECT 1, 2,20,0  
MAKE OBJECT CONE 2,8  
POSITION OBJECT 2,0,5,0  
MAKE OBJECT PLAIN 4,100,100  
LOAD IMAGE "grid8by8.bmp",1  
TEXTURE OBJECT 4,1  
XROTATE OBJECT 4,-90
```

continued on next page

LISTING-48.4
(continued)

Adding a Realistic ODE
Volume for the Cone

```
REM *** Wait to start ODE ***  
WAIT KEY  
  
ODE START  
  
REM *** Build ODE volumes ***  
ODE CREATE DYNAMIC BOX 1  
ODE CREATE STATIC BOX 2  
ODE CREATE STATIC BOX 4  
  
REM *** Continually update physics ***  
DO  
    ODE UPDATE  
LOOP  
  
REM *** End program ***  
ODE END  
END  
  
FUNCTION ScreenSetUp()  
    SET DISPLAY MODE 1280,1024,32  
    COLOR BACKDROP 0  
    BACKDROP ON  
    AUTOCAM OFF  
    POSITION CAMERA 0,10,-50  
    POINT CAMERA 0,0,0  
ENDFUNCTION
```

Activity 48.10

Type in and test the program in LISTING-48.4 (*ODE04.dbpro*).

As we see, the cube stops when it hits the cone. It doesn't tumble, because the cone is using a box collision volume. A more accurate result would be obtained if a static mesh was used on the cone. This is done using the ODE CREATE STATIC TRIANGLE MESH statement which has the format shown in FIG-48.12.

FIG-48.12

The ODE CREATE
STATIC TRIANGLE
MESH Statement



In the diagram:

objno is an integer value giving the ID of the 3D object to which the ODE mesh is to be linked.

Activity 48.11

In your last program, change the ODE volume for object 2 to use a static mesh.
How does this affect the result?

The ODE SET WORLD ERP Statement

ERP stands for **Error Reduction Parameter**.

When two objects collide, the physics engine handles this by creating a temporary link (known as a **joint**) between the two objects. This joint is used to calculate how the two objects should react to the collision.

In performing this calculation some rounding errors will occur. How these errors

are handled can be set using the ODE SET WORLD ERP statement which has the format shown in FIG-48.13.

FIG-48.13

The ODE SET WORLD
ERP Statement



In the diagram:

value

is a real number which should lie between 0 and 1. A value of zero will result in no corrective adjustments; 1 will give maximum correction.

For example, we could switch off corrective action using the line:

```
ODE SET WORLD ERP 0.0
```

In fact, this statement can have a major effect on how objects react to collisions. In the next Activity we'll see just what effects it can have on the cube as it falls onto the cone.

Activity 48.12

In your last program, add the line

```
ODE SET WORLD ERP 0.0
```

immediately after the ODE START statement.

How does this affect the falling cube?

Try running the program again with first a value of 1.0 and then 0.5 for the ERP.

The ODE SET WORLD CFM Statement

CFM stands for
**Constraint Force
Mixing.**

Falling onto a concrete base is going to hurt a lot more than falling onto a trampoline - it's all to do with bounce. The more give there is in a material the more things bounce. We can set the bounce factor for all the dynamic objects in our ODE universe using the ODE SET WORLD CFM statement which has the format shown in FIG-48.14.

FIG-48.14

The ODE SET WORLD
CFM Statement



In the diagram:

value

is a real number which should lie between 0 and 5. A value of zero gives a very rigid (no bounce) effect; 5 gives maximum bounce.

For example, a large bounce can be effected using the line:

```
ODE SET WORLD CFM 5
```

Activity 48.13

Add the line

```
ODE SET WORLD CFM 2.0
```

immediately after the ERP statement in your last program.

The ODE SET CONTACT FDIR1 Statement

FDIR1 stands for Friction DIRection 1 (there's a second friction direction).

FIG-48.15

The ODE SET CONTACT FDIR1 Statement

You should have seen how the cube slides along the surface after it lands. This is due to a lack of friction - as if the block was made of melting ice. We can set the friction of a dynamic object using the ODE SET CONTACT FDIR1 statement which has the format shown in FIG-48.15.



In the diagram:

objno

is an integer value giving the ID of the object to which a friction coefficient is to be applied.

friction

is a real number setting the friction coefficient of the object. A value of zero results in no friction (the default for all objects) while 100 gives maximum friction.

To add a friction coefficient to our falling cube we could use the line:

```
ODE SET CONTACT FDIR1 1,50
```

Activity 48.14

Add the above statement to your last program. Insert the line immediately after the ODE volume is assigned to the cube.

The ODE SET LINEAR VELOCITY Statement

We can add a velocity to an object using the ODE SET LINEAR VELOCITY statement which has the format shown in FIG-48.16.

FIG-48.16

The ODE SET LINEAR VELOCITY Statement



In the diagram:

objno

is an integer value giving the ID of the object to which a velocity is to be applied.

xcomp

is a real value giving the value of the x component of the velocity.

ycomp

is a real value giving the value of the y component of the velocity.

zcomp

is a real value giving the value of the z component of the velocity.

For example, we could make object 1 move horizontally using the line:

```
ODE SET LINEAR VELOCITY 1, 20,0,0
```

A larger value will result in faster movement.

The program in LISTING-48.5 moves a cube from left to right above the ground plane. Notice that gravitational forces are set to zero.

LISTING-48.5

Giving an Object a Velocity

```
ScreenSetUp()
REM *** Make objects ***
MAKE OBJECT CUBE 1,7
POSITION OBJECT 1, -20,20,0
MAKE OBJECT PLAIN 4,100,100
LOAD IMAGE "grid8by8.bmp",1
TEXTURE OBJECT 4,1
XROTATE OBJECT 4,-90
REM *** Wait to start ODE ***
WAIT KEY
ODE START
REM *** Build ODE volumes ***
ODE CREATE DYNAMIC BOX 1
ODE CREATE STATIC BOX 4
REM *** No gravity ***
ODE SET WORLD GRAVITY 0,0,0
REM *** Add a velocity to the cube ***
ODE SET LINEAR VELOCITY 1,20,0,0
REM *** Continually update physics ***
DO
  ODE UPDATE
LOOP
REM *** End program ***
ODE END
END

FUNCTION ScreenSetUp()
  SET DISPLAY MODE 1280,1024,32
  COLOR BACKDROP 0
  BACKDROP ON
  AUTOCAM OFF
  POSITION CAMERA 0,10,-50
  POINT CAMERA 0,0,0
ENDFUNCTION
```

Activity 48.15

Type in and test the program given in LISTING-48.5 (*ODE05.dbpro*).

Find out the effect of each of the following velocity settings:

<i>xcomp</i>	<i>ycomp</i>	<i>zcomp</i>
100	0	0
-10	0	0
0	-5	0
0	0	20
0	-10	20

If we switch gravity back on, this will affect the path taken by the object to which a velocity has been applied.

Activity 48.16

Return the velocity settings of the cube to 0,0,20.

Remove the ODE SET GRAVITY statement, thereby returning the gravitational force to its default setting.

Run the program again. How is the cube's path affected?

In the next program (see LISTING-48.6), two cubes travel towards each other and collide.

LISTING-48.6

Colliding Bodies

```
ScreenSetUp()

REM *** Make cubes ***
MAKE OBJECT CUBE 1,7
POSITION OBJECT 1, -20,20,0
MAKE OBJECT CUBE 2,7
POSITION OBJECT 2,20,20,0

REM *** Wait to start ODE ***
WAIT KEY
ODE START

REM *** Build ODE volumes ***
ODE CREATE DYNAMIC BOX 1
ODE CREATE DYNAMIC BOX 2

REM *** No gravity ***
ODE SET WORLD GRAVITY 0,0,0

REM *** Add a velocity to the cubes ***
ODE SET LINEAR VELOCITY 1,2,0,0
ODE SET LINEAR VELOCITY 2,-2,0,0
REM *** Continually update physics ***
DO
    ODE UPDATE
LOOP

REM *** End program ***
ODE END
END

FUNCTION ScreenSetUp()
    SET DISPLAY MODE 1280,1024,32
    COLOR BACKDROP 0
    BACKDROP ON
    AUTOCAM OFF
    POSITION CAMERA 0,10,-70
    POINT CAMERA 0,0,0
ENDFUNCTION
```

Activity 48.17

Type in and test the program given in LISTING-48.6 (*ODE06.dbpro*).

What happens when the two cubes collide?

The force applied to each cube is equal but opposite, so the collision results in these energies cancelling each other out. Of course, if the velocities of the two cubes were to be different, then the result obtained would change.

Activity 48.18

In your last program, what effect is achieved for each of the following settings in the ODE SET LINEAR VELOCITY statement for object 1:

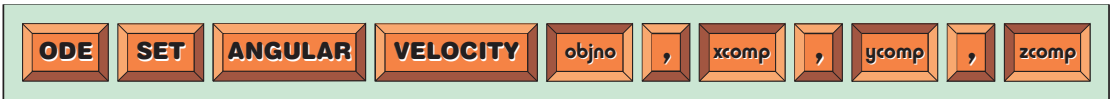
<i>xcomp</i>	<i>ycomp</i>	<i>zcomp</i>
4	0	0
4	-1	0

The ODE SET ANGULAR VELOCITY Statement

We can set an object spinning by assigning it an angular velocity. This is achieved using the ODE SET ANGULAR VELOCITY statement which has the format shown in FIG-48.17.

FIG-48.17

The ODE SET ANGULAR VELOCITY Statement



In the diagram:

- objno* is an integer value giving the ID of the object to which an angular velocity is to be applied.
- xcomp* is a real value giving the spin force about the x-axis.
- ycomp* is a real value giving the spin force about the y-axis.
- zcomp* is a real value giving the spin force about the z-axis.

To keep the spin to a reasonably slow rate, *xcomp*, *ycomp*, and *zcomp* values should be less than 1.

For example, to create a slow spin of object 1 with rotation about the x-axis being half the speed of rotation about the y-axis, we could use the line:

```
ODE SET ANGULAR VELOCITY 1, 0.1,0.2,0
```

Activity 48.19

Modify your last program by including the statement given above immediately after the linear velocity of object 1 has been set.

How is the cube affected?

Move the ODE SET ANGULAR VELOCITY statement placing it within the program's DO..LOOP structure.

The ODE SET BODY ROTATION Statement

It is also possible to get an object to rotate to a specific position about its own local axes using the ODE SET BODY ROTATION statement. Although we already have a ROTATE OBJECT statement, it is important to use the ODE version of the statement if we want the ODE box (sphere, cylinder, etc.) associated with the object also to be rotated.

FIG-48.18

The ODE SET BODY ROTATION statement has the format shown in FIG-48.18.

The ODE SET BODY ROTATION Statement



In the diagram:

- objno* is an integer value giving the ID of the object to which the rotation is to be applied.
- xangle* is a real value giving the object's angle of rotation about its x-axis.
- yangle* is a real value giving the object's angle of rotation about its y-axis.
- zangle* is a real value giving the object's angle of rotation about its z-axis.

For example, we could rotate object 1 45° about its y-axis using the statement:

```
ODE SET BODY ROTATION 1,0,45,0
```

Activity 48.20

Replace the ODE SET ANGULAR VELOCITY statement in your last program with the ODE SET BODY ROTATION statement given above.

Notice that the cube is set to a specific angle and is then fixed. Since this statement creates a single movement of the object, it need not be within the DO..LOOP structure.

Move the ODE SET BODY ROTATION statement to a position before the loop structure and check that the overall effect remains the same.

The ODE SET BODY MASS Statement

In the real world, objects have weight - or, rather, mass. In ODE objects only have a mass if they are explicitly assigned one. This can be done using the ODE SET BODY MASS statement which has the format given in FIG-48.19.

FIG-48.19

The ODE SET BODY MASS Statement



In the diagram:

objno

is an integer value giving the ID of the object to which a mass is to be assigned.

value

is a real value giving the mass to be assigned to the object. This value does not represent any specific units of measurement.

Activity 48.21

Modify your last program so that the mass of object 1 is set to 8 and that of object 2 to 100.

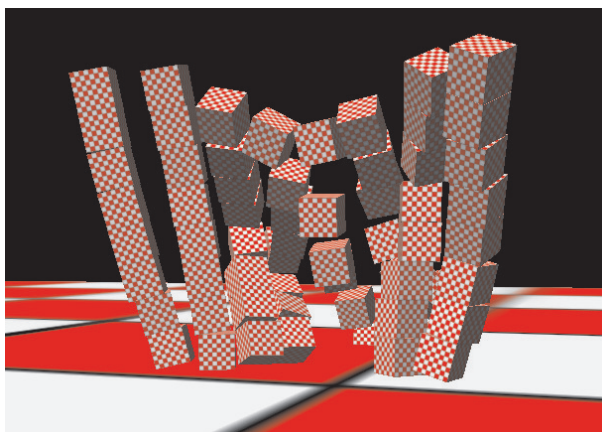
How does this effect the cubes?

Set gravity to -0.1. How are the cubes affected this time?

The program in LISTING-48.7 shows the effects produced by throwing a ball at a wall of cubes (see FIG-48.20).

FIG-48.20

Tumbling Blocks



LISTING-48.7

Tumbling Blocks

```
ScreenSetUp()

REM *** Load image used to texture objects ***
LOAD IMAGE "grid8by8.bmp",1

REM *** Make, texture and position cubes ***
objno = 1
FOR row = 0 TO 7
  FOR col = 0 TO 7
    MAKE OBJECT CUBE objno,7
    TEXTURE OBJECT objno,1
    POSITION OBJECT objno,col*7-28,row*7+6,0
    INC objno
  NEXT col
NEXT row

REM *** Make ball
MAKE OBJECT SPHERE 65,10
POSITION OBJECT 65,0,30,-100

REM *** Make ground ***
MAKE OBJECT PLAIN 66,1000,1000
TEXTURE OBJECT 66,1
XROTATE OBJECT 66,-90
```

continued on next page

LISTING-48.7

(continued)

Tumbling Blocks

```
REM *** Start ODE ***
ODE START

REM *** Static box around floor ***
ODE CREATE STATIC BOX 66

REM *** Build ODE volumes around cubes ***
objno = 1
FOR row = 0 TO 7
  FOR col = 0 TO 7
    ODE CREATE DYNAMIC BOX objno
    ODE SET CONTACT FDIR1 objno,80
    INC objno
  NEXT col
NEXT row

REM *** ODE volume around sphere ***
ODE CREATE DYNAMIC SPHERE 65

REM *** Give sphere a push ***
ODE SET LINEAR VELOCITY 65,0,0,30

REM *** Engage physics after key press ***
WAIT KEY
DO
  ODE UPDATE
LOOP

REM *** End program ***
ODE END
END

FUNCTION ScreenSetUp()
  SET DISPLAY MODE 1280,1024,32
  COLOR BACKDROP 0
  BACKDROP ON
  AUTOCAM OFF
  POSITION CAMERA 50,25,-150
  POINT CAMERA 0,0,0
ENDFUNCTION
```

Activity 48.22

Type in and test the program in LISTING-48.7 (*ODE07.dbpro*).

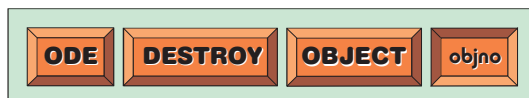
Modify the program so that the camera can be moved about within the scene.
(HINT: include the *camera.dba* routines we created in Chapter 33.)

The ODE DESTROY OBJECT Statement

The ODE object linked to a regular 3D object can be deleted using the ODE DESTROY OBJECT statement which has the format shown in FIG-48.21.

FIG-48.21

The ODE DESTROY
OBJECT Statement



In the diagram:

objno

is an integer value giving the ID of the 3D object to which the ODE volume is attached.

The ODE object attached to the specified 3D object is deleted; the 3D object itself is not affected by the operation, but its movement is no longer controlled by the physics engine.

Activity 48.23

Inside the DO..LOOP structure of your last program, add the following lines:

```
IF SPACEKEY ()
    ODE DESTROY OBJECT 65
ENDIF
```

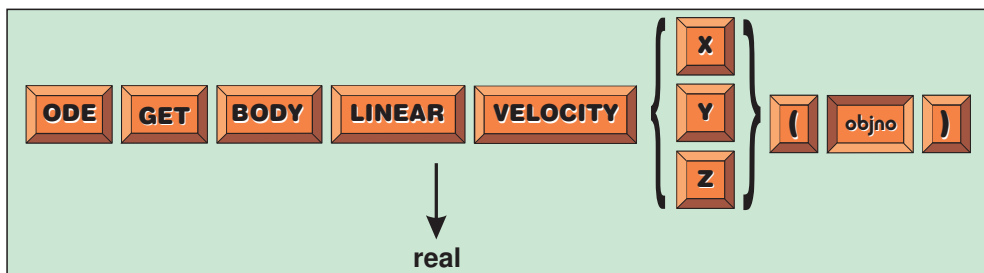
This will destroy the ODE volume attached to the sphere.

Test the new program by destroying the ODE volume before the ball hits the blocks. How is the program affected?

The ODE GET BODY LINEAR VELOCITY Statement

We can discover the x, y and z components of an object's velocity using the ODE GET BODY LINEAR VELOCITY statement which has the format shown in FIG-48.22.

FIG-48.22 The ODE GET BODY LINEAR VELOCITY Statement



In the diagram:

X,Y,Z

Use X to retrieve the x component of the velocity;
use Y to retrieve the y component of the velocity;
use Z to retrieve the z component of the velocity.

objno

is an integer value giving the ID of the object
whose velocity is to be determined.

Activity 48.24

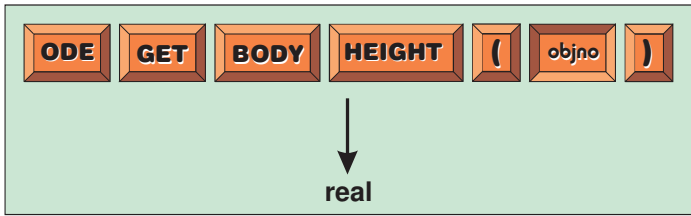
Modify the program in *ODE06.dbpro* to display both objects' linear velocity at all times.

The ODE GET BODY HEIGHT Statement

The ODE GET BODY HEIGHT returns the height of a specified object. Although we might be tempted to think that an object will have whatever height we set it to when the object is first created, an irregularly shaped object will change in height as it is rotated. The format for this statement is given in FIG-48.23.

FIG-48.23

The ODE GET BODY
HEIGHT Statement



In the diagram:

objno

is an integer value giving the ID of the object whose height is to be determined.

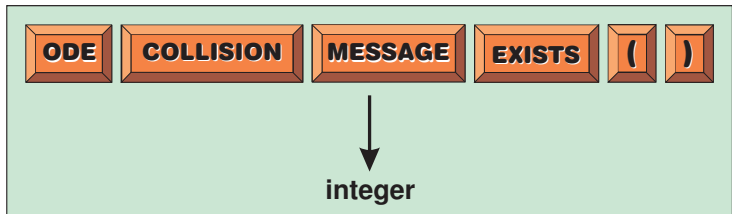
The statement returns the length of the object along the world y-axis.

The ODE COLLISION MESSAGE EXISTS Statement

When a collision occurs, details of that collision are placed on a stack. We can check if that stack currently contains any messages using the ODE COLLISION MESSAGE EXISTS statement which has the format shown in FIG-48.24.

FIG-48.24

The ODE COLLISION
MESSAGE EXISTS
Statement



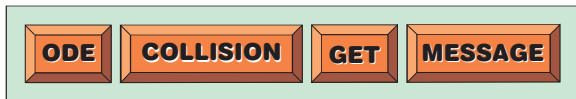
The statement returns 1 if at least one message exists in the stack; otherwise zero is returned.

The ODE COLLISION GET MESSAGE Statement

The top message in the ODE collision stack can be retrieved using the ODE COLLISION GET MESSAGE statement (format shown in FIG-48.25).

FIG-48.25

The ODE COLLISION
GET MESSAGE
Statement



The details in the value retrieved from the stack can then be accessed using other commands.

Typical code for retrieving collision details from the stack would be:

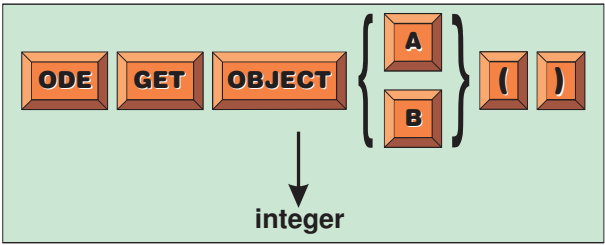
```
IF ODE COLLISION MESSAGE EXISTS ()
  ODE COLLISION GET MESSAGE
ENDIF
```

The ODE GET OBJECT Statement

To discover what two objects have been in collision, we need to use the ODE GET OBJECT statement which takes the format shown in FIG-48.26.

FIG-48.26

The ODE GET OBJECT
Statement



In the diagram:

A,B Use A to retrieve the ID of the first object involved in a collision; use B to retrieve the ID of the second object.

This statement should be used after a message has been retrieved from the collision stack.

For example, we could see which objects have collided using the code:

```
IF ODE COLLISION MESSAGE EXISTS()  
  ODE COLLISION GET MESSAGE  
  obj1 = ODE GET OBJECT A()  
  obj2 = ODE GET OBJECT B()  
  REPEAT  
    SET CURSOR 100,100  
    PRINT "Objects collided: ",obj1," ",obj2  
  UNTIL INKEY$() <> ""  
ENDIF
```

Activity 48.25

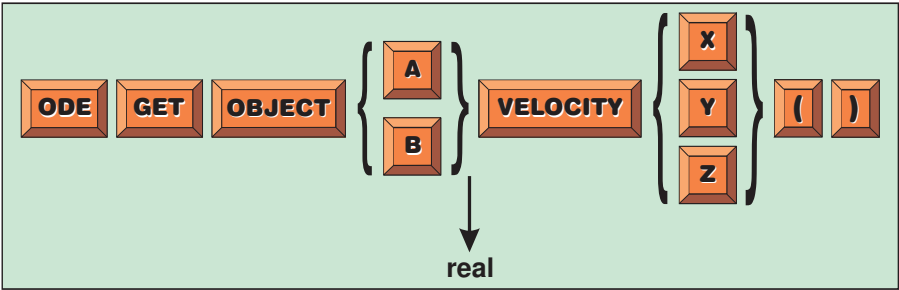
Reload *ODE06.dbpro*. Create a function, *DisplayCollision()*, from the lines above and modify the DO..LOOP to display only the output from this function. Run the program and check that the objects colliding are identified.

The ODE GET OBJECT VELOCITY Statement

The x, y and z components of a colliding object's velocity can be determined using the ODE GET OBJECT VELOCITY statement whose format is shown in FIG-48.27.

FIG-48.27

The ODE GET
OBJECT
VELOCITY
Statement



In the diagram:

A,B Use A to retrieve the ID of the first object involved in a collision; use B to retrieve the ID of the second object.

X,Y,Z

Use X to retrieve the x component of the velocity;
use Y to retrieve the y component of the velocity;
use Z to retrieve the z component of the velocity.

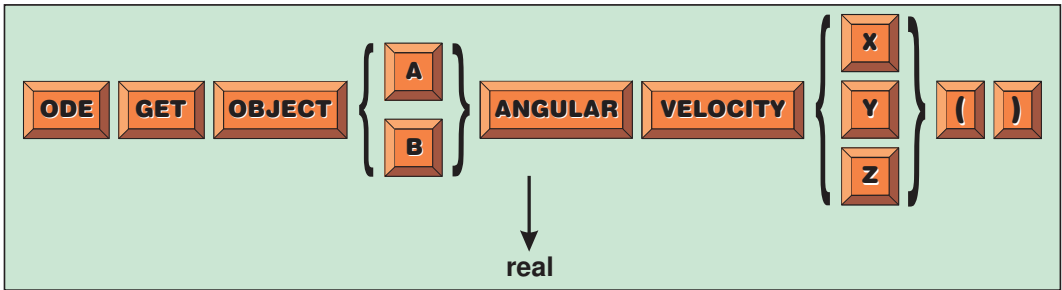
Activity 48.26

Update *DisplayCollision()* in your last program to display the velocity offsets of each colliding object.

The ODE GET OBJECT ANGULAR VELOCITY Statement

The x, y and z components of a colliding object's angular velocity can be determined using the ODE GET OBJECT ANGULAR VELOCITY statement whose format is shown in FIG-48.28.

FIG-48.28 The ODE GET OBJECT ANGULAR VELOCITY Statement



In the diagram:

A,B

Use A to retrieve the ID of the first object involved in a collision; use B to retrieve the ID of the second object.

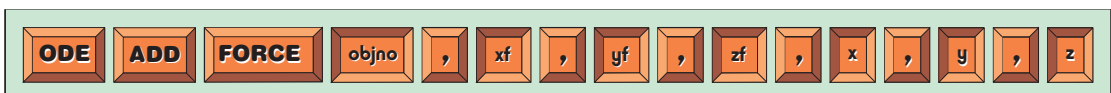
X,Y,Z

Use X to retrieve the x component of the angular velocity; use Y to retrieve the y component of the angular velocity; use Z to retrieve the z component of the angular velocity.

The ODE ADD FORCE Statement

An additional force can be applied to an object using the ODE ADD FORCE statement. The statement allows the magnitude and direction of the force to be specified as well as the point on the object at which the force is to be applied. By applying the force off-centre, we add a spin to the object. The format for the ODE ADD FORCE statement is given in FIG-48.29.

FIG-48.29 The ODE ADD FORCE Statement



In the diagram:

objno

is an integer value giving the ID of the object to which the force is to be applied.

x, y, z

are a set of real values giving the strength of the force along each axis.

 x, y, z

are a set of real values giving the position on the object at which the force is to be applied.

The program in LISTING-48.8 allows the user to apply a force to a cube by pressing the space bar.

LISTING-48.8

Applying a Force

```
ScreenSetUp()

REM *** Force to be applied ***
force# = 0.01

REM *** Force already applied indicator ***
applied = 0

REM *** Start ODE and eliminate gravity ***
ODE START
ODE SET WORLD GRAVITY 0,0,0

REM *** Load dice model and add mass ***
LOAD OBJECT "dice.dbo",2
ODE CREATE DYNAMIC BOX 2
ODE SET BODY MASS 2, 5

REM *** Engage physics engine ***
DO
    ODE UPDATE

    REM *** First time space bar pressed, apply force ***
    IF SPACEKEY()=1 AND (NOT applied)
        ODE ADD FORCE 2,force#,0,0,0,0
        REM *** Remember force has been applied ***
        applied=1
    ENDIF
LOOP

REM *** End program ***
ODE END
END

FUNCTION ScreenSetUp()
    SET DISPLAY MODE 1280,1024,32
    COLOR BACKDROP RGB(125,125,125)
    BACKDROP ON
    AUTOCAM OFF
    POSITION CAMERA 0,10,-150
    POINT CAMERA 0,0,0
ENDFUNCTION
```

Activity 48.27

Type in and test the program in LISTING-48.8 (*ODE08.dbpro*).

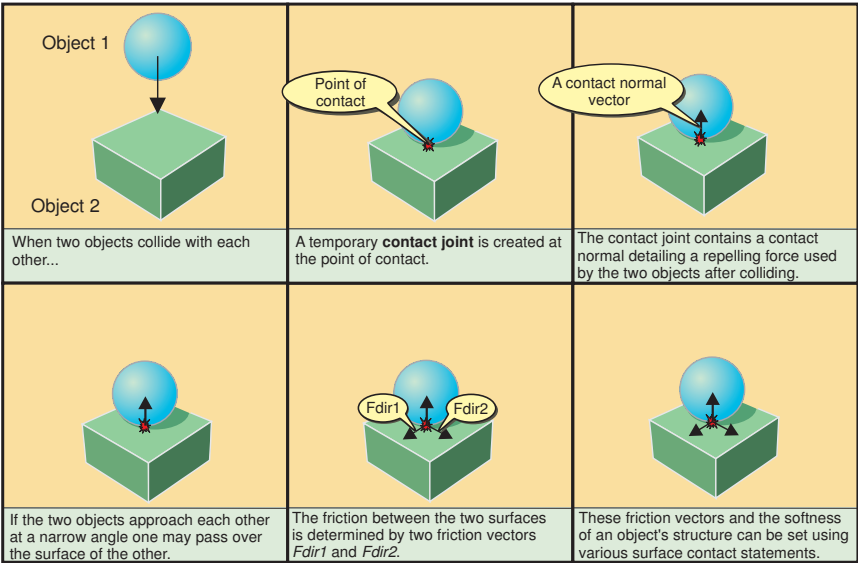
Try modifying the value of *force#* and applying it to the *y* and *z* directions instead of *x*.

Check out the effects of changing the point at which the force is applied.

Surface Contact Statements

When two objects collide, a **contact joint** is created. This contact joint contains details of how the two objects should react to that collision (see FIG-48.30).

FIG-48.30 A Contact Joint



There are a series of statements allowing the programmer to fine-tune how the collision should be dealt with and the forces involved. We can set factors such as the *bounciness* of the objects, and the surface friction components along both directions (*Fdir1* and *Fdir2*).

Exactly which options are to be used must first be selected using a set of ODE SETSURFACE MODE CONTACT statements which are designed to select/deselect various characteristics. These statements and a brief explanation of what effect is being selected are shown in TABLE-48.1.

TABLE-48.1

SETSURFACE
CONTACT MODE
Statements

Statement	Parameters	Description
ode set surface mode contact bounce	objno:int, state:int	When <i>state</i> = 0, <i>objno</i> , will create a bouncye collision
ode set surface mode contact soft erp	objno:int, state:int	When <i>state</i> = 0, <i>objno</i> 's ERP can be set. Useful for creating soft objects.
ode set surface mode contact soft cfm	objno:int, state:int	When <i>state</i> = 0, <i>objno</i> 's CFM can be set. Useful when creating soft objects.
ode set surface mode contact motion1	objno:int, state:int	When <i>state</i> = 0, <i>objno</i> 's surface is assumed to be moving independently (like a conveyer belt) in direction <i>Fdir1</i>
ode set surface mode contact motion2	objno:int, state:int	When <i>state</i> = 0, <i>objno</i> 's surface is assumed to be moving independently in direction <i>Fdir2</i>
ode set surface mode contact slip1	objno:int, state:int	When <i>state</i> = 0, a force-dependent slip in direction <i>Fdir1</i> can be applied.
ode set surface mode contact slip2	objno:int, state:int	When <i>state</i> = 0, a force-dependent slip in direction <i>Fdir2</i> can be applied.
ode set surface mode contact approx11	objno:int, state:int	When <i>state</i> = 0 calculate friction using a friction-pyramid approximation in direction <i>Fdir1</i>
ode set surface mode contact approx12	objno:int, state:int	When <i>state</i> = 0 calculate friction using a friction-pyramid approximation in direction <i>Fdir2</i>
ode set surface mode contact approx1	objno:int, state:int	When <i>state</i> = 0 calculate friction using a friction-pyramid approximation in direction <i>Fdir1</i> and <i>Fdir2</i>

Once the required options have been selected, the strengths of actual forces involved can be set using various ODE SET CONTACT statements. These are described briefly in TABLE-48.2.

TABLE-48.2

ODE SET CONTACT
Statements

Statement	Parameters	Description
ode set contact fdir 1	objno: int, value:real	Sets the degree of friction along <i>Fdir1</i> on <i>objno</i> to <i>value</i> . <i>value</i> range 0 to 100. 0 = no friction 100 = max friction
ode set contact mu2	objno: int, value:real	Sets the degree of friction along <i>Fdir2</i>
ode set contact bounce	objno: int, value:real	Sets the amount of elasticity and bounce of <i>objno</i> to <i>value</i> . <i>value</i> range 0 to 1.
ode set contact velocity	objno: int, value:real	Sets minimum velocity required to create a bounce effect.
ode set contact soft erp	objno: int, value:real	Sets ERP for <i>objno</i> to <i>value</i>
ode set contact soft cfm	objno: int, value:real	Sets CPM for <i>objno</i> to <i>value</i>
ode set contact motion1	objno: int, value:real	Sets the movement factor along <i>Fdir1</i> for <i>objno</i> to <i>value</i>
ode set contact motion2	objno: int, value:real	Sets the movement factor along <i>Fdir2</i> for <i>objno</i> to <i>value</i>
ode set contact slip1	objno: int, value:real	Sets the slip value to be applied along direction <i>Fdir1</i> to <i>value</i>
ode set contact slip2	objno: int, value:real	Sets the slip value to be applied along direction <i>Fdir1</i> to <i>value</i>

The program in LISTING-48.9 demonstrates the bounce effect, bouncing a falling dice off the ground. All the statements necessary to create a bounce effect are held in a separate function, *CreateBounce()*. Notice that the effects are applied only to the dynamic object.

LISTING-48.9

Creating Bounce

```
ScreenSetUp()

REM *** Start ODE ***
ODE START
ODE SET WORLD STEP 0.05
ODE SET WORLD ERP (0.2)*2.5
ODE SET WORLD CFM (10^-5)*2.5
ODE SET STEP MODE 0

REM *** Make ground ***
MAKE OBJECT BOX 1,500,10,500
LOAD IMAGE "grid8by8.bmp",1
TEXTURE OBJECT 1,1

REM *** Load dice ***
LOAD OBJECT "dice.dbo",2
POSITION OBJECT 2,-10,100,0

REM *** Add ODE components to objects ***
ODE CREATE STATIC BOX 1
ODE CREATE DYNAMIC BOX 2

REM ** Give the dice weight ***
ODE SET BODY MASS 2,100

CreateBounce(2)

DO
    ODE UPDATE
LOOP

REM *** End program ***
ODE END
END
```

continued on next page

LISTING-48.9

(continued)

Creating Bounce

```
FUNCTION ScreenSetUp()  
    SET DISPLAY MODE 1280,1024,32  
    AUTOCAM OFF  
    POSITION CAMERA 0,30,-150  
ENDFUNCTION  
  
FUNCTION CreateBounce(objno)  
    ODE SETSURFACE MODE CONTACT BOUNCE objno,0  
    ODE SET CONTACT BOUNCE objno,0.7  
    ODE SETSURFACE MODE CONTACT SOFT ERP objno,0  
    ODE SET CONTACT SOFT ERP objno, 0.8  
    ODE SETSURFACE MODE CONTACT SOFT CFM objno, 0  
    ODE SET CONTACT SOFT CFM objno,200.5  
ENDFUNCTION
```

Activity 48.28

Type in and test the program in LISTING-48.9 (*ODE09.dbpro*).

Try varying the parameters of the statements in the new function and check out what effect this has.

The next routine tests out the conveyer-belt effect created using the CONTACT MOTION1 and CONTACT MOTION2 statements:

```
FUNCTION ConveyerBelt(objno)  
    ODE SET CONTACT FDIR1 objno,50  
    `ODE SETSURFACE MODE CONTACT MOTION1 objno,0  
    `ODE SET CONTACT MOTION1 objno,10  
    `ODE SETSURFACE MODE CONTACT MOTION2 objno,0  
    `ODE SET CONTACT MOTION2 objno,10  
ENDFUNCTION
```

Activity 48.29

Add the code for *ConveyerBelt()* to your last program (notice that most of the lines have been commented out at this stage).

In the main section of your program, comment out the call to *CreateBounce()* and replace it with the lines:

```
ODE ADD FORCE 2,4,0,0,0,0,0  
ConveyerBelt(2)
```

Run the program and observe how the dice moves.

Now remove the comments from lines 2 and 3 in *ConveyerBelt()* and run the program again. What happens to the dice this time?

Next, re-comment out lines 2 and 3 and uncomment lines 4 and 5. How does the dice react ?

Remove all comments in *ConveyerBelt()* and run the program again. What path does the dice take?

Summary

- The Open Dynamics Engine (ODE) is an undocumented part of DarkBASIC Pro.
- By using ODE we can make basic laws of physics apply to 3D objects.
- ODE places invisible volumes around 3D objects.
- Static volumes should be placed around 3D objects which are stationery.
- Dynamic volumes should be placed around objects which can be moved.
- Use ODE CREATE DYNAMIC BOX to place a box volume around a moveable object.
- Use ODE CREATE DYNAMIC SPHERE to place a spherical volume around a moveable object.
- Use ODE CREATE DYNAMIC CYLINDER to place a cylindrical volume around a moveable object.
- Use ODE CREATE DYNAMIC TRIANGLE MESH to place a mesh-based volume around a moveable object.
- Use ODE CREATE STATIC BOX to place a box volume around an immovable object.
- Use CREATE STATIC TRIANGLE MESH to place a mesh-based volume around an immovable object.
- Use ODE START before any other ODE statements in your program. This initialises the ODE.
- Use ODE END to end use of the ODE.
- Use ODE UPDATE in a loop structure to continually update the calculations and movements performed by the ODE.
- By default, ODE assumes a gravitational pull down the y-axis.
- Use ODE SET WORLD GRAVITY to modify the gravitational force used in a program.
- Use ODE SET WORLD STEP to set the time interval assumed to have passed between each ODE update.
- Use ODE SET WORLD ERP to specify the error production parameter.
- Use ODE SET WORLD CFM to set the bounciness of every ODE object.
- Use ODE SET CONTACT FDIR1 to set the surface friction of an object.
- Use ODE SET LINEAR VELOCITY to set the velocity of an object.
- Use ODE SET ANGULAR VELOCITY to set the angular velocity of an object.

- Use ODE SET BODY ROTATION to rotate an object to a specific angle.
- Use ODE SET BODY MASS to assign a mass to an object.
- Use ODE DESTROY OBJECT to destroy the ODE volume surrounding a standard 3D object.
- Use ODE GET BODY LINEAR VELOCITY to determine the linear velocity of an object.
- Use ODE GET BODY HEIGHT to determine the height of an object. This may change as an asymmetrical shape is rotated.
- Use ODE COLLISION MESSAGE EXISTS to determine if a collision has taken place between two objects and the corresponding details placed on a collision stack.
- Use ODE COLLISION GET MESSAGE to retrieve the latest entry in the collision stack.
- Use ODE GET OBJECT to determine which two objects have collided. This statement is used after an entry has been retrieved from the collision stack.
- Use ODE GET OBJECT VELOCITY to retrieve the velocities of the two objects that have collided. This statement is used after an entry has been retrieved from the collision stack.
- Use ODE GET OBJECT ANGULAR VELOCITY to retrieve the angular velocities of the two objects that have collided. This statement is used after an entry has been retrieved from the collision stack.
- Use ODE ADD FORCE to apply a force to a specified object.
- Surface contact statements are used to set the characteristics of a collision.

Solutions

Activity 48.1

No solution required.

Activity 48.2

The cube fall varies in speed and direction with each setting.

Activity 48.3

The code is:

```
ScreenSetUp()  
REM *** Make cube ***  
MAKE OBJECT CUBE 1,10  
POSITION OBJECT 1,0,25,0  
REM *** Make ground ***  
MAKE OBJECT PLAIN 2, 100,100  
XROTATE OBJECT 2,90  
WAIT KEY  
REM *** Start Physics engine ***  
ODE START  
ODE SET WORLD GRAVITY 0,-2,0  
REM *** Link ODE box to cube ***  
ODE CREATE DYNAMIC BOX 1  
REM *** Update screen ***  
DO  
    ODE UPDATE  
LOOP  
REM *** End program ***  
ODE END  
END
```

The cube passes straight through the ground plane.

Activity 48.4

The code is:

```
ScreenSetUp()  
REM *** Make cube ***  
MAKE OBJECT CUBE 1,10  
POSITION OBJECT 1,0,25,0  
REM *** Make ground ***  
MAKE OBJECT PLAIN 2, 100,100  
XROTATE OBJECT 2,90  
WAIT KEY  
REM *** Start Physics engine ***  
ODE START  
ODE SET WORLD GRAVITY 0,-2,0  
REM *** Link ODE box to cube and plane ***  
ODE CREATE DYNAMIC BOX 1  
ODE CREATE DYNAMIC BOX 2  
REM *** Update screen ***  
DO  
    ODE UPDATE  
LOOP  
REM *** End program ***  
ODE END  
END
```

The plane which is meant to represent ground level also falls!

Activity 48.5

Change the line

```
ODE CREATE DYNAMIC BOX 2
```

to

```
ODE CREATE STATIC BOX 2
```

The ground stays still and the cube stops when it hits the ground.

Change

```
ODE CREATE DYNAMIC BOX 1
```

to

```
ODE CREATE DYNAMIC BOX 1,20,20,20
```

With a larger surrounding volume, the cube stops well above ground level.

Activity 48.6

Since a sphere is not an accurate enclosure for a cube, the cube again stops above ground level.

Activity 48.7

No solution required.

Activity 48.8

Neither volume seems to be very accurate.

Activity 48.9

The whole process runs much more slowly at 0.05. Using 0.2 speeds things up.

Activity 48.10

No solution required.

Activity 48.11

Change

```
ODE CREATE STATIC BOX 2
```

to

```
ODE CREATE STATIC TRIANGLE MESH 2
```

Although it has a static volume, the cone sinks into the ground! The cube tilts on hitting the top of the cone.

Activity 48.12

The ERP values have the following effects:

- 0.0 the cube slides
- 1.0 the cube bounces
- 0.5 the cube tilts over the cone

Activity 48.13

This change adds a slight bounce before the cube slides off.

Activity 48.14

This stops the cube sliding off along the ground.

Activity 48.15

The effects are:

100,0,0	cube moves quickly to the right
-10,0,0	cube moves slowly to the right
0,5,0	cube moves up
0,0,20	cube moves off into the screen
0,-10,20	cube moves down and in, then moves in only

Activity 48.16

As the cube moves away it also falls, eventually hitting the ground and sliding on until it falls over the edge of the ground.

Activity 48.17

The cubes stop moving when they collide.

Activity 48.18

When object 1's velocity is set to 4,0,0, it pushes object 2 to the right.

When object 1's velocity is set to 4,-1,0, it tilts and halts object 2.

Activity 48.19

Object 1 tilts slightly, but does not continue to rotate.

When the statement is moved within the DO..LOOP, the cube continues to rotate as it moves.

Activity 48.20

No solution required.

Activity 48.21

The final version of the main section of the program should now be:

```
ScreenSetUp()  
REM *** Make cubes ***  
MAKE OBJECT CUBE 1,7  
POSITION OBJECT 1, -20,20,0  
MAKE OBJECT CUBE 2,7  
POSITION OBJECT 2,20,20,0  
REM *** Wait to start ODE ***  
WAIT KEY  
ODE START  
REM *** Build ODE volumes ***  
ODE CREATE DYNAMIC BOX 1  
ODE CREATE DYNAMIC BOX 2
```

```
REM *** Slight gravity ***  
ODE SET WORLD GRAVITY 0,-0.1,0  
REM *** Set cubes' characteristics ***  
ODE SET LINEAR VELOCITY 1,4,-1,0  
ODE SET BODY MASS 1, 8  
ODE SET BODY ROTATION 1,0,45,0  
ODE SET LINEAR VELOCITY 2,-2,0,0  
ODE SET BODY MASS 2, 100  
REM *** Continually update physics ***  
DO  
    ODE UPDATE  
LOOP  
REM *** End program ***  
ODE END  
END
```

Since object 2 has a much larger mass, it has a greater effect on object 1's trajectory.

With a slight gravity, the cubes fall as they move.

Activity 48.22

To move the camera about the scene, the DO..LOOP structure should be changed to:

```
DO  
    ODE UPDATE  
    PointCameraUsingMouse(0)  
    MoveCameraUsingMouse(0,0.5)  
LOOP
```

Also, to include the necessary code, we need to add the line

```
#INCLUDE "camera.dba"
```

at the start of the program.

Activity 48.23

The main loop should now be:

```
DO  
    ODE UPDATE  
    IF SPACEKEY()  
        ODE DESTROY OBJECT 65  
    ENDIF  
    PointCameraUsingMouse(0)  
    MoveCameraUsingMouse(0,0.5)  
LOOP
```

When the ODE volume is destroyed, the sphere is no longer influenced by the physics engine and stops moving.

Activity 48.24

To keep the main section of the program simple, it is probably best if we do the hard work in two routines:

```
FUNCTION DisplayVelocities()  
    SET CURSOR 100,100  
    PRINT "Velocity (object 1):",  
        %VelocityDetails$(1)  
    SET CURSOR 100,130  
    PRINT "Velocity (object 2):",  
        %VelocityDetails$(2)  
ENDFUNCTION  
  
FUNCTION VelocityDetails$(objno)  
    x# = ODE GET BODY LINEAR VELOCITY  
        %X(objno)  
    y# = ODE GET BODY LINEAR VELOCITY
```

```

    ↵ Y(objno)
    z# = ODE GET BODY LINEAR VELOCITY
    ↵ Z(objno)
    result$ = "X: "+STR$(x#)+" Y: "+STR$(y#)
    ↵ +" Z: "+STR$(z#)
ENDFUNCTION result$

```

The DO..LOOP only needs one new line:

```

DO
    ODE UPDATE
    DisplayVelocities()
LOOP

```

Activity 48.25

The code for the function is:

```

FUNCTION DisplayCollisions()
    IF ODE COLLISION MESSAGE EXISTS()
        ODE COLLISION GET MESSAGE
        obj1 = ODE GET OBJECT A()
        obj2 = ODE GET OBJECT B()
        REPEAT
            SET CURSOR 100,100
            PRINT "Objects collided: ",obj1,
                ↵ " ",obj2
            UNTIL INKEY$() <> ""
        ENDIF
    ENDIF
ENDFUNCTION

```

The DO..LOOP should now be:

```

DO
    ODE UPDATE
    DisplayCollisions()
LOOP

```

Activity 48.26

Again, we start by adding a helper function to perform all the work

```

FUNCTION CollisionVelocityDetails$(op$)
    IF op$ = "A"
        x# = ODE GET OBJECT A VELOCITY X()
        y# = ODE GET OBJECT A VELOCITY Y()
        z# = ODE GET OBJECT A VELOCITY Z()
        result$ = "X: "+STR$(x#)+" Y: "
            ↵ "+STR$(y#)+" Z: "+STR$(z#)
    ELSE
        x# = ODE GET OBJECT B VELOCITY X()
        y# = ODE GET OBJECT B VELOCITY Y()
        z# = ODE GET OBJECT B VELOCITY Z()
        result$ = "X: "+STR$(x#)+" Y: "
            ↵ "+STR$(y#)+" Z: "+STR$(z#)
    ENDIF
ENDFUNCTION result$

```

and then updating *DisplayCollisions()* to print the extra details:

```

FUNCTION DisplayCollisions()
    IF ODE COLLISION MESSAGE EXISTS()
        ODE COLLISION GET MESSAGE
        obj1 = ODE GET OBJECT A()
        obj2 = ODE GET OBJECT B()
        REPEAT
            SET CURSOR 100,100
            PRINT "Objects collided: ",obj1,
                ↵ " ",obj2
            SET CURSOR 100,130
            PRINT "Collision velocity "

```

```

            ↵ ,obj1,"": ",
            ↵ CollisionVelocityDetails$("A")
            SET CURSOR 100,160
            PRINT "Collision velocity "
            ↵ ,obj2,"": ",
            ↵ CollisionVelocityDetails$("B")
        UNTIL INKEY$() <> ""
    ENDIF
ENDFUNCTION

```

Activity 48.27

By increasing the value of the force, the cube speeds up. By changing the direction, the cube moves in another direction. By moving the position at which the force is applied, the cube turns about one or more axes.

Activity 48.28

No solution required.

Activity 48.29

On the first run, the dice slides off to the right.

On the second run, with lines 2 and 3 of *ConveyerBelt()* included, the cube slides off to the left.

On the third run, with lines 4 and 5 of *ConveyerBelt()* included, the cube slides off into the screen.

On the last run, with lines 2, 3, 4 and 5 of *ConveyerBelt()* included, the cube reverts to sliding off to the left.

Vectors and Matrices

3D Vectors

4D Vectors

Matrices for Shaders

Introduction

Back in Chapter 25 we looked at how 2D vectors could be used when manipulating sprites. To perform the same sort of manipulations on a 3D object we need to employ 3D vectors. Many of the operations performed on 3D vectors are identical in nature to those employed in 2D vectors. But since the basic concepts are covered in Volume 1, which not everyone may have, we'll quickly run over the terminology.

A Mathematical Description of 3D Vectors

Whereas a single value (such as 12, or -3) is known as a **scalar**, the term **vector** is used to denote a list of values. The values within a vector are termed the **elements** of the vector and the number of elements determines the vector's **dimensions**. So a vector containing three elements is a three dimensional vector. Vectors can be shown horizontally, [12.3,4.6,1.7] and are known as **row vectors**, or vertically

$$\begin{bmatrix} 12.3 \\ 4.6 \\ 1.7 \end{bmatrix}$$

in which case they are termed **column vectors**.

A vector is often assigned an identifying name, normally a lowercase letter shown in bold. For example,

$$\mathbf{a} = [12.3, 4.6, 1.7]$$

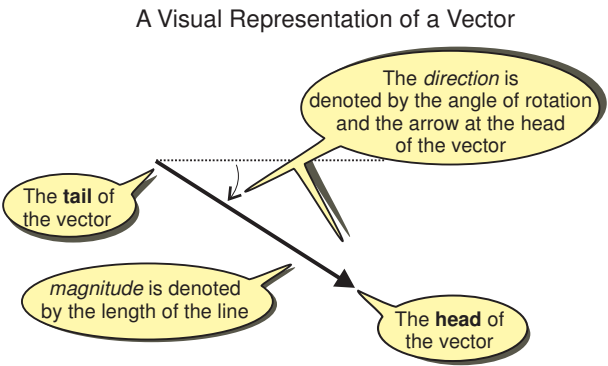
The individual elements within a 3D vector are named x, y and z. To refer to an element with a named vector, we use the vector name followed by the element name in subscript. For example,

$$\mathbf{a}_x = 12.3$$

In a graphical context, the term vector refers to a directed line that has both **magnitude** and **direction** (see FIG-49.1).

FIG-49.1

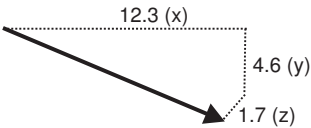
Visualisation of a Vector



The details of a vector can be stored as three numbers giving the position of the head of the vector relative to its tail, as shown in FIG-49.2.

FIG-49.2

How a 3D Vector is Specified



From the offset values held within the vector, its magnitude can be determined. For vector **a**, its magnitude (written as $\|a\|$) can be calculated using the formula:

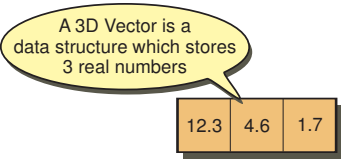
$$\|a\| = \sqrt{a_x^2 + a_y^2 + a_z^2}$$

What is a 3D Vector in DarkBASIC Pro?

A 3D vector is nothing more than a record structure capable of holding three real numbers (see FIG-49.3). What makes it a little different is that DarkBASIC Pro contains a great number of commands for manipulating 3D vectors.

FIG-49.3

A 3D Vector

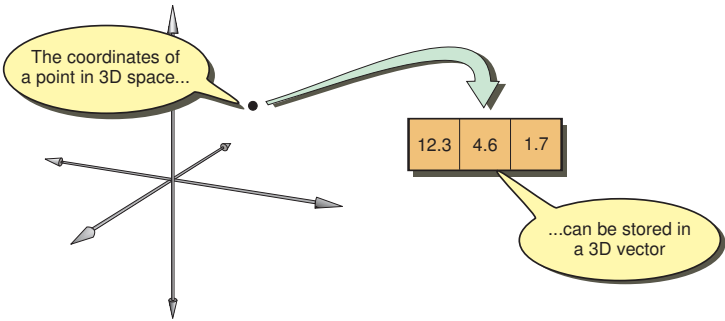


Why do we need 3D Vectors?

A 3D vector is simply a convenient storage area for the coordinates of a point in 3D space (see FIG-49.4).

FIG-49.4

Storing a Point's Coordinates in a 3D Vector



It can also be used to store an object's velocity.

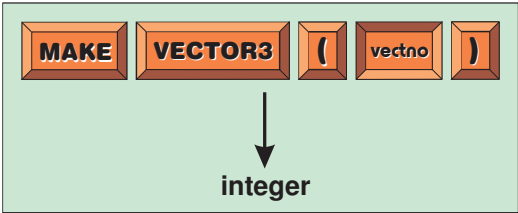
3D Vector Statements

The MAKE VECTOR3 Statement

To create a 3D vector we need to use the MAKE VECTOR3 statement which has the format shown in FIG-49.5.

FIG-49.5

The MAKE VECTOR3 Statement



In the diagram:

vectorno

is an integer value which is to be the ID of the vector being created.

This statement returns 1 if the vector is created successfully; otherwise zero is returned. Normally, we wouldn't bother checking the returned value, so we'd create a 3D vector with a statement such as:

```
MAKE VECTOR3 (1)
```

To help identify vectors within a program it's probably best, in all but the shortest programs, to use a constant for the vector ID:

```
#CONSTANT myvector  
MAKE VECTOR3 (myvector)
```

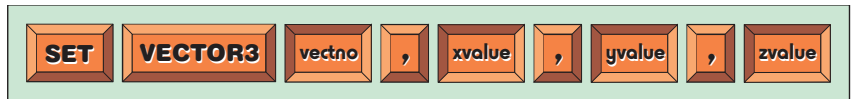
The fields within a 3D vector are identified as *x*, *y* and *z*.

The SET VECTOR3 Statement

To assign values to the fields within a 3D vector, we use the SET VECTOR3 statement which has the format shown in FIG-49.6.

FIG-49.6

The SET VECTOR3
Statement



In the diagram:

vectorno

is an integer value giving the ID of the 3D vector to be assigned a value.

xvalue

is a real number specifying the value to be assigned to the *x* field within the 3D vector.

yvalue

is a real number specifying the value to be assigned to the *y* field within the 3D vector.

zvalue

is a real number specifying the value to be assigned to the *z* field within the 3D vector.

For example, we could assign the values 12.3, 4.6 and 1.7 to *myvector* using the lines:

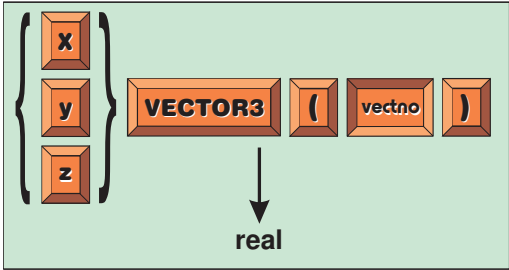
```
#CONSTANT myvector  
MAKE VECTOR3 (myvector)  
SET VECTOR3 myvector, 12.3, 4.6, 1.7
```

Retrieving Data from a 3D Vector

To retrieve one of the values stored within a 3D vector, we use the VECTOR3 statement which has the format shown in FIG-49.7.

FIG-49.7

The VECTOR3 Statement



In the diagram:

- X,Y,Z

Use the option appropriate for the value you wish to retrieve.
- vectno*

is an integer value giving the ID of the vector whose data is to be accessed.

For example, using the 3D vector we set up in the example above, the line

```
PRINT Y VECTOR3(myvector)
```

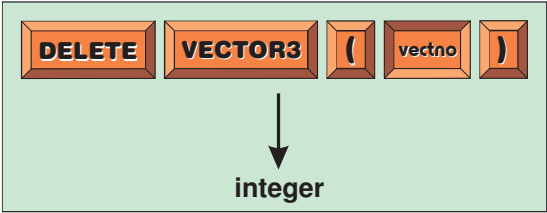
would display the value 4.6.

The DELETE VECTOR3 Statement

When finished with a 3D vector, we can delete it from RAM using the DELETE VECTOR3 statement which has the format shown in FIG-49.8.

FIG-49.8

The DELETE VECTOR3 Statement



In the diagram:

- vectno*

is an integer value giving the ID of the vector to be deleted.

If the vector is successfully deleted, this statement returns 1; otherwise zero is returned.

The COPY VECTOR3 Statement

The contents of one 3D vector can be copied to another using the COPY VECTOR3 statement which has the format shown in FIG-49.9.

FIG-49.9

The COPY VECTOR3 Statement



In the diagram:

- destvectno*

is an integer value giving the ID of the destination 3D vector.

srcvectno

is an integer value giving the ID of the source 3D vector.

For example, we could copy the contents of vector 2 to vector 1 using the line

```
COPY VECTOR3 1,2
```

The MULTIPLY VECTOR3 Statement

Each value within a 3D vector can be multiplied by a specified factor using the MULTIPLY VECTOR3 statement which has the format shown in FIG-49.10.

FIG-49.10

The MULTIPLE VECTOR3 Statement



In the diagram:

vectno

is an integer value giving the ID of the 3D vector which is to be used in the calculation.

multiplier

is a real number giving the value by which the elements of the vector are to be multiplied.

The contents of *vectno* are modified by this operation.

For example, 3D vector 1 could have its contents multiplied by 1.3 using the line:

```
MULTIPLE VECTOR3 1, 1.3
```

The SCALE VECTOR3 Statement

The SCALE VECTOR3 statement is similar to MULTIPLE VECTOR3 but this time the result is stored in a different vector and the original vector left unchanged. The format for the SCALE VECTOR3 statement is given in FIG-49.11.

FIG-49.11

The SCALE VECTOR3 Statement



In the diagram:

destvectno

is an integer value giving the ID of the 3D vector in which the result is to be stored.

srcvectno

is an integer value giving the ID of the vector whose contents are to be used in the calculation.

multiplier

is a real number giving the value by which the elements of the vector are to be multiplied.

The contents of *srcvectno* are not altered by this operation.

We could use this statement to store the result of multiplying the contents of vector 2 by 1.3 in vector 1 using the line:

```
SCALE VECTOR3 1,2,1.3
```

The DIVIDE VECTOR3 Statement

We can divide a vector's contents by a specified amount using the DIVIDE VECTOR3 statement whose format is shown in FIG-49.12.

FIG-49.12

The DIVIDE VECTOR3 Statement



In the diagram:

- vectno* is an integer value giving the ID of the vector which is to be used in the operation.
- divisor* is a real number giving the value by which the contents of the vector are to be divided.

The contents of *vectno* are altered by this operation.

To divide the contents of 3D vector 1 by 2, we would use the line:

```
DIVIDE VECTOR3 1,2
```

Activity 49.1

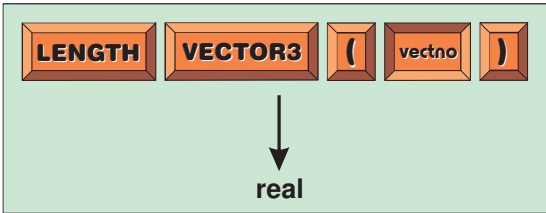
Write a program (*act4901.dbpro*) which reads in 3 real numbers from the keyboard and stores them in a 3D vector. When the up-arrow key is pressed, the contents of the vector should be increased by 10% and when the down-arrow key is pressed, the vector's contents should be decreased by 10%. Display the vector's contents each time it is changed.

The LENGTH VECTOR3 Statement

If a 3D vector is being used to represent direction and speed, the speed is represented by the magnitude of the vector. That magnitude can be calculated from the details held within a 3D vector using the LENGTH VECTOR3 statement whose format is shown in FIG-49.13.

FIG-49.13

The LENGTH VECTOR3 Statement



In the diagram:

- vectno* is an integer value giving the ID of the vector whose magnitude is to be determined.

Activity 49.2

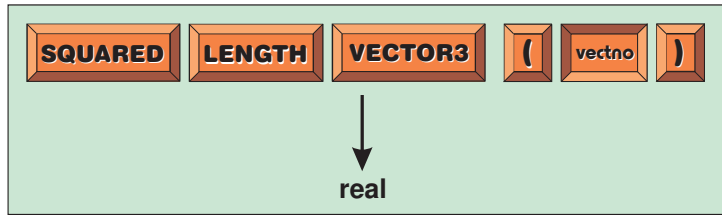
Add to your previous program so that the vector is used to set the velocity of a cube. The speed of the cube should be displayed.

The SQUARED LENGTH VECTOR3 Statement

While LENGTH VECTOR3 returns the magnitude of a vector, the square of the magnitude can be discovered using the SQUARED LENGTH VECTOR3 statement which has the format shown in FIG-49.14.

FIG-49.14

The SQUARED
LENGTH VECTOR3
Statement



In the diagram:

vectno

is an integer value giving the ID of the vector for which the square of the magnitude is to be determined.

The ADD VECTOR3 Statement

We can add two 3D vectors using the ADD VECTOR3 statement. The format for this statement is shown in FIG-49.15.

FIG-49.15

The ADD VECTOR3
Statement



In the diagram:

destvectno

is an integer value giving the ID of the vector which is to contain the result of the calculation.

vectno1

is an integer value giving the ID of the first vector involved in the calculation.

vectno2

is an integer value giving the ID of the second vector involved in the calculation.

For example, the statements

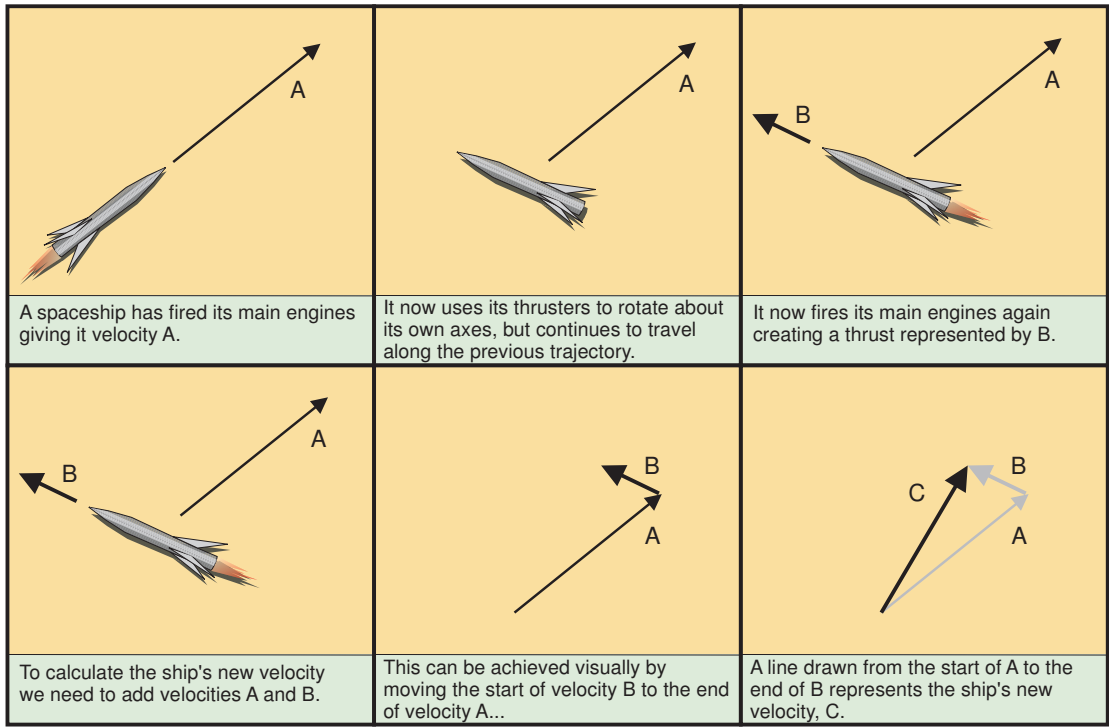
```
#CONSTANT  v1      1
#CONSTANT  v2      2
#CONSTANT  v3      3
ADD VECTOR3 v3,v1,v2
```

will result in the following:

```
v3.x = v1.x + v2.x
v3.y = v1.y + v2.y
v3.z = v1.z + v2.z
```

The obvious question at this point is; Why might we want to add two vectors? Simply, if two thrust vectors are added, it gives us the resulting thrust (see FIG-49.16).

FIG-49.16 Why we need to Add Vectors



The SUBTRACT VECTOR3 Statement

We can subtract the contents of one vector from another and store the result in a third using the SUBTRACT VECTOR3 statement which has the format shown in FIG-49.17.

FIG-49.17

The SUBTRACT VECTOR3 Statement



In the diagram:

- destvectno* is an integer value giving the ID of the vector which is to contain the result of the calculation.
- vectno1* is an integer value giving the ID of the first vector involved in the calculation.
- vectno2* is an integer value giving the ID of the second vector involved in the calculation.

If we assume vectors **a** and **b** contain coordinates of two points rather than velocities, then the SUBTRACT VECTOR3 statement can be used to calculate the position of point *b* relative to point *a* by subtracting vector **a** from vector **b**.

The DOT PRODUCT VECTOR3 Statement

When two vectors, **a** and **b**, are multiplied, this is written as **a.b**, the dot being used

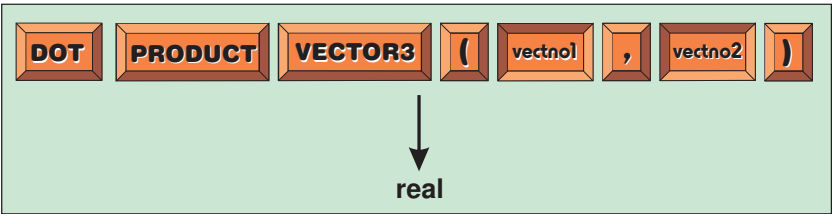
to indicate multiplication. The calculation performed by this operation is:

$$a_x b_x + a_y b_y + a_z b_z$$

We can perform this operation in DarkBASIC Pro using the DOT PRODUCT VECTOR3 statement which has the format shown in FIG-49.18.

FIG-49.18

The DOT PRODUCT VECTOR3 Statement



In the diagram:

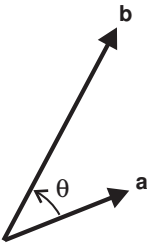
vectno1, *vectno2*

are integer values giving the IDs of the vectors involved in the multiplication.

This operation is used when we need to discover the angle between two vectors (see FIG-49.19).

FIG-49.19

The Angle Between Two Vectors



The angle θ (theta) between vectors **a** and **b** can be calculated using the formula:

$$\theta = \arccos\left(\frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\| \|\mathbf{b}\|}\right)$$

The NORMALIZE VECTOR3 Statement

The formula for calculating the angle between two vectors becomes simpler if both vectors **a** and **b** have a magnitude of 1, the formula then reducing to:

$$\theta = \arccos(\mathbf{a} \cdot \mathbf{b})$$

It is possible to reduce a vector's magnitude to 1 without affecting its direction using the NORMALIZE VECTOR3 statement which has the format shown in FIG-49.20.

FIG-49.20

The NORMALIZE VECTOR3 Statement



In the diagram:

destvectno

is an integer value giving the ID of the destination 3D vector.

srcvectno

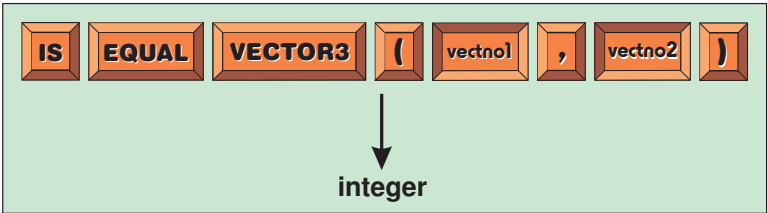
is an integer value giving the ID of the source 3D vector.

The IS EQUAL VECTOR3 Statement

We can check if two 3D vectors contain identical values using the IS EQUAL VECTOR3 statement which has the format shown in FIG-49.21.

FIG-49.21

The IS EQUAL VECTOR3 Statement



In the diagram:

vectno1, *vectno2* are integer values giving the IDs of the two vectors being compared.

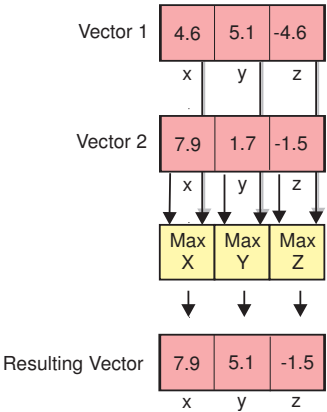
If the contents of the two vectors are identical, the statement returns 1; otherwise zero is returned.

The MAXIMIZE VECTOR3 Statement

We can construct a vector using the maximum values from two existing vectors with the MAXIMIZE VECTOR3 statement. The concept behind the statement is demonstrated visually in FIG-49.22.

FIG-49.22

How MAXIMIZE VECTOR3 Operates



The format for the MAXIMIZE VECTOR3 statement is given in FIG-49.23.

FIG-49.23

The MAXIMIZE VECTOR3 Statement



In the diagram:

destvectno is an integer value giving the ID of the vector which is to contain the result of the calculation.

vectno1 is an integer value giving the ID of the first vector involved in the calculation.

vectno2 is an integer value giving the ID of the second vector involved in the calculation.

The MINIMIZE VECTOR3 Statement

A new vector constructed from the minimum values from two other vectors can be constructed using the MINIMIZE VECTOR3 statement whose format is given in FIG-49.24.

FIG-49.24

The MINIMIZE
VECTOR3 Statement



In the diagram:

destvectno

is an integer value giving the ID of the vector which is to contain the result of the calculation.

vectno1

is an integer value giving the ID of the first vector involved in the calculation.

vectno2

is an integer value giving the ID of the second vector involved in the calculation.

The CROSS PRODUCT VECTOR3 Statement

We can determine the expression for a vector which is at right-angles to two existing 3D vectors using the CROSS PRODUCT VECTOR3 statement which has the format shown in FIG-49.25.

FIG-49.25 The CROSS PRODUCT VECTOR3 Statement



In the diagram:

destvectno

is an integer value giving the ID of the vector which is to contain the result of the calculation.

vectno1

is an integer value giving the ID of the first vector involved in the calculation.

vectno2

is an integer value giving the ID of the second vector involved in the calculation.

For example, using the vectors [0,0,100] and [0,10,0] (which are parallel to the z and y axes respectively) the CROSS PRODUCT VECTOR3 statement would create a new vector at right angles to these - that is a vector parallel to the x-axis. This is demonstrated in LISTING-49.1.

LISTING-49.1

Creating a Cross Product
Vector

```
SET DISPLAY MODE 1280,1024,32
r = MAKE VECTOR3 (1)
SET VECTOR3 1, 0,0,100
r = MAKE VECTOR3 (2)
SET VECTOR3 2, 0,10,0
r = MAKE VECTOR3 (3)
CROSS PRODUCT VECTOR3 3,1,2
x# = X VECTOR3 (3)
```

continued on next page

LISTING-49.1

(continued)

Creating a Cross Product
Vector

```
y# = Y VECTOR3 (3)
z# = Z VECTOR3 (3)
PRINT "VECTOR 3", x#, " ", y#, " ", z#
WAIT KEY
END
```

Activity 49.3

Type in and test the program given in LISTING-49.1 (*vector01.dbpro*).

Summary

- 3D vectors are structures which hold 3 real numbers.
- The fields within a vector object are identified as x, y, and z.
- A 3D vector can be used to store the coordinates of a point in 3D space or the velocity of an object in 3D space.
- Use MAKE VECTOR3 to create a 3D vector object.
- Use SET VECTOR3 to assign values to a 3D vector.
- Use X VECTOR3 to retrieve the x value within a 3D vector. Use Y VECTOR3 and Z VECTOR3 to retrieve the other values.
- Use DELETE VECTOR3 to delete a 3D vector from RAM.
- Use COPY VECTOR3 to copy the contents of one vector into another.
- Use MULTIPLY VECTOR3 to multiply each element within a vector by a fixed value. The result is stored in the named vector.
- Use SCALE VECTOR3 to multiply each element within a vector by a fixed amount and store the result in a second vector.
- Use DIVIDE VECTOR3 to divide the elements of a vector by a fixed value. The result is stored in the named vector.
- Use LENGTH VECTOR3 to calculate the magnitude of a velocity vector. This is only appropriate when the vector is used to store a velocity rather than a point's coordinates.
- Use SQUARED LENGTH VECTOR3 to calculate the square of the magnitude of a velocity.
- Use ADD VECTOR3 to add the contents of two vectors and store the result in a third vector.
- Use SUBTRACT VECTOR3 to subtract the contents of one vector from another and store the result in a third vector.
- Use DOT PRODUCT VECTOR3 to calculate the dot product of two vectors.
- Use NORMALIZE VECTOR3 to calculate a vector with a magnitude of 1 but pointing in the same direction as an existing vector.

- Use `IS EQUAL VECTOR3` to check if two vectors are exactly equal.
- Use `MAXIMIZE VECTOR3` to create a new vector from the maximum values in two other vectors.
- Use `MINIMIZE VECTOR3` to create a new vector from the minimum values in two other vectors.
- Use `CROSS PRODUCT VECTOR3` to create a vector which is at right angles to two existing vectors.

4D Vectors

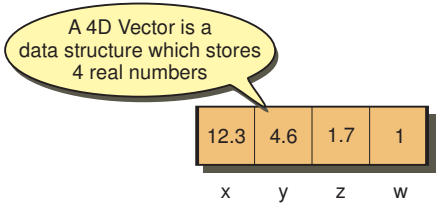
Introduction

Sorry, but DarkBASIC Pro doesn't allow you to create 4-dimensional objects - despite the title of this section. But we do need vectors which contain 4 elements.

These vectors are used to store 3D coordinates or vectors, but a fourth element (w) is added (see FIG-49.26) because some of the mathematics we need to perform demand 4 element vectors, not 3.

FIG-49.26

A 4D Vector



The new element within the vector is known as w . We are free to assign any value we wish to w .

To convert the contents of a 3D vector, \mathbf{a} , to the equivalent 4D in vector \mathbf{b} , the following formula is used:

$$\begin{aligned}b_x &= a_x * w \\b_y &= a_y * w \\b_z &= a_z * w \\b_w &= w\end{aligned}$$

To convert from 4D to 3D the formula is:

$$\begin{aligned}a_x &= b_x / w \\a_y &= b_y / w \\a_z &= b_z / w\end{aligned}$$

These calculations become trivial if w is set to 1.

Coordinates stored in this fashion, with an extra w component, are known as **homogeneous coordinates**.

Activity 49.4

Convert the point (30,9,-12) to homogeneous coordinates in a 4D vector ($w=1$).

Convert the homogeneous vector [12, 3, -18, 2] to the equivalent 3D vector.

Many of the commands available for 4D vectors perform the same operations as those already covered for 2 and 3 dimensional vectors, so rather than describe each one separately they are listed below in TABLE-49.1.

TABLE-49.1 4D Vector Statements

Statement	Parameters	Returned	Description
ADD VECTOR4	VR, V1, V2		VR set to $V1 + V2$
COPY VECTOR4	VR, V		VR set to V
DELETE VECTOR4	V	num(0 or 1)	deletes V returns 1 if OK
DIVIDE VECTOR4	VR, num		divides VR by num
IS EQUAL VECTOR4	V1, V2	num (0 or 1)	returns 1 if $V1 = V2$
LENGTH VECTOR4	V	len	len set magnitude of V
MAKE VECTOR4	V	num (0 or 1)	creates V; returns 1 if OK
MAXIMIZE VECTOR4	VR, V1, V2		VR set to max values from V1 and V2
MINIMIZE VECTOR4	VR, V1, V2		VR set to min values from V1 and V2
MULTIPLY VECTOR4	VR, num		multiplies VR by num
NORMALIZE VECTOR4	VR,V		VR set to normalised version of V
SCALE VECTOR4	VR, V, num		VR set to $V*num$
SET VECTOR4	VR, xn, yn, zn, wn		VR set to $[xn,yn,zn,wn]$
SQUARED LENGTH VECTOR4	V	len	len set to magnitude of V squared
SUBTRACT VECTOR4	VR, V1, V2		VR set to $V1 - V2$
W VECTOR4	V	num	num set to V_w
X VECTOR4	V	num	num set to V_x
Y VECTOR4	V	num	num set to V_y
Z VECTOR4	V	num	num set to V_z

Introduction

Don't confuse the term matrix as it is used here with the matrix grids we created back in Chapter 45. Here we use the term matrix to mean a grid of values - just like a two-dimensional array. An example of a matrix is shown below:

$$\begin{bmatrix} 4 & 10 & 8 \\ 7 & -42 & 0 \end{bmatrix}$$

The matrix in the example is a 2 by 3 matrix (2 rows by 3 columns).

Like vectors, matrices are often assigned names, usually shown in uppercase bold:

$$\mathbf{M} = \begin{bmatrix} 4 & 10 & 8 \\ 7 & -42 & 0 \end{bmatrix}$$

Just like arrays, the individual elements within a matrix are identified by their row and column positions. Some texts start subscripting from 1, hence the top-left element is referred to as 1,1; others start subscripts at 0 and the top-left element is then 0,0. We will subscript from 0,0 since DarkBASIC Pro arrays begin subscripting at zero.

$$\mathbf{M} = \begin{bmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \end{bmatrix}$$

DarkBASIC Pro has statements for creating and manipulating only 4 by 4 matrices.

The matrices in DarkBASIC Pro are really intended for use with shaders (see Chapter 50) and hence have very limited accessibility. They can be created and manipulated but are not designed to be examined in any way.

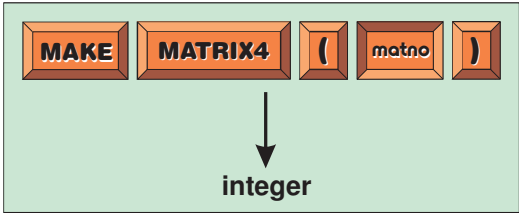
Matrix Statements

The MAKE MATRIX4 Statement

To create a 4 by 4 matrix we use the MAKE MATRIX4 statement which has the format shown in FIG-49.27.

FIG-49.27

The MAKE MATRIX4
Statement



In the diagram:

matno

is an integer value giving the ID to be assigned to the matrix being created.

The statement returns 1 if the matrix was created successfully; otherwise zero is returned.

You could be forgiven for thinking you can now set up any values you want within the matrix you've just created, but, in fact, DarkBASIC Pro matrix objects are designed to hold only specific values which are loaded into the matrix using tailor-made statements.

The SET IDENTITY MATRIX4 Statement

When dealing with scalar values, the number 1 is a rather important value; it is the only number that has no effect when we multiply or divide by it. A similar value exists for matrices and is called the **identity matrix**. The 4 by 4 identity matrix is:

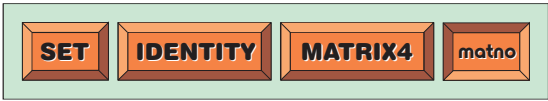
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Notice the diagonal line of 1s - this is the sign of an identity matrix, no matter what size the matrix is, although it must be square.

We can set the contents of a matrix to the identity matrix using the SET IDENTITY MATRIX4 statement which has the format shown in FIG-49.28.

FIG-49.28

The SET IDENTITY MATRIX4 Statement



In the diagram:

matno

is an integer value specifying the matrix to be set to the identity matrix value.

A typical statement would be:

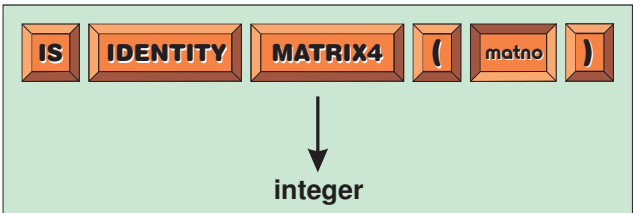
SET IDENTITY MATRIX4 1

The IS IDENTITY MATRIX4 Statement

We can check to find out if a matrix contains the identity matrix using the IS IDENTITY MATRIX4 statement whose format is shown in FIG-49.29.

FIG-49.29

The IS IDENTITY MATRIX4 Statement



In the diagram:

matno

is an integer value specifying the matrix to be tested.

Other Matrix Assignment Statements

As a 3D object's details are sent to your graphics card, it is possible to pick up the coordinates of each polygon and store these in a 4D matrix. The coordinates can be accessed in various formats: world coordinates (the position of vertices in 3D space), view coordinates (coordinates within the viewport), or projection coordinates (coordinates on screen). A statement exists to load each of these options (see TABLE-49.2).

TABLE-49.2

Other Matrix
Assignment Statements

Statement	Parameters	Returned	Description
PROJECTION MATRIX4	MR		MR set to projection values
VIEW MATRIX4	MR		MR set to view values
WORLD MATRIX4	MR		MR set to world values

For example, we could load the world coordinates of a polygon into matrix 1 using the line:

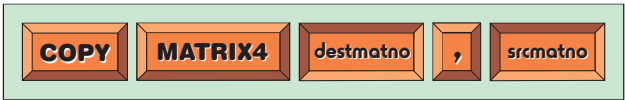
WORLD MATRIX4 1

The COPY MATRIX4 Statement

The contents of one matrix can be copied to another using the COPY MATRIX4 statement which has the format shown in FIG-49.30.

FIG-49.30

The COPY MATRIX4
Statement



In the diagram:

destmatno

is an integer value giving the ID of the destination 4D matrix.

srcmatno

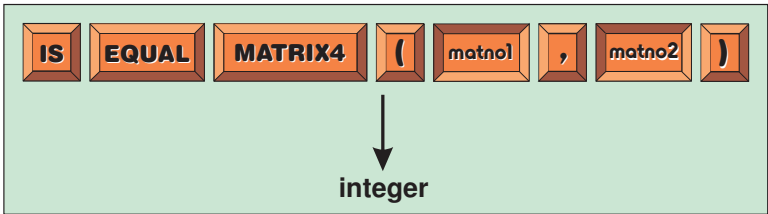
is an integer value giving the ID of the source 4D matrix.

The IS EQUAL MATRIX4 Statement

To check if the contents of two matrices are exactly equal, we can use the IS EQUAL MATRIX4 statement which has the format shown in FIG-49.31.

FIG-49.31

The IS EQUAL
MATRIX4 Statement



In the diagram:

matno1, *matno2*

are integer values giving the IDs of the two matrices being compared.

If the contents of the two matrices are identical, the statement returns 1; otherwise zero is returned.

The ADD MATRIX4 Statement

The contents of two 4 by 4 matrices can be added using the ADD MATRIX4 statement which has the format shown in FIG-49.32.

FIG-49.32

The ADD MATRIX4 Statement



In the diagram:

destmatno is an integer value giving the ID of the matrix which is to contain the result of the calculation.

matno1 is an integer value giving the ID of the first matrix involved in the calculation.

matno2 is an integer value giving the ID of the second matrix involved in the calculation.

Two 4 by 4 matrices are added using the formula shown below:

$$\mathbf{A} = \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} b_{00} & b_{01} & b_{02} & b_{03} \\ b_{10} & b_{11} & b_{12} & b_{13} \\ b_{20} & b_{21} & b_{22} & b_{23} \\ b_{30} & b_{31} & b_{32} & b_{33} \end{bmatrix} \quad \mathbf{A+B} = \begin{bmatrix} a_{00}+b_{00} & a_{01}+b_{01} & a_{02}+b_{02} & a_{03}+b_{03} \\ a_{10}+b_{10} & a_{11}+b_{11} & a_{12}+b_{12} & a_{13}+b_{13} \\ a_{20}+b_{20} & a_{21}+b_{21} & a_{22}+b_{22} & a_{23}+b_{23} \\ a_{30}+b_{30} & a_{31}+b_{31} & a_{32}+b_{32} & a_{33}+b_{33} \end{bmatrix}$$

The SUBTRACT MATRIX4 Statement

The contents of two 4 by 4 matrices can be subtracted using the SUBTRACT MATRIX4 statement which has the format shown in FIG-49.33.

FIG-49.33

The SUBTRACT MATRIX4 Statement



In the diagram:

destmatno is an integer value giving the ID of the matrix which is to contain the result of the calculation.

matno1 is an integer value giving the ID of the first matrix involved in the calculation.

matno2 is an integer value giving the ID of the second matrix involved in the calculation.

The DIVIDE MATRIX4 Statement

We can divide the contents of a matrix by a scalar value using the DIVIDE MATRIX4 statement whose format is shown in FIG-49.34.

FIG-49.34

The DIVIDE MATRIX4 Statement



In the diagram:

- destmatno*

is an integer value giving the ID of the matrix which is to contain the result of the calculation.
- divisor*

is a real value. Each element within the matrix will have its original value divided by this amount.

The MULTIPLY MATRIX4 Statement

We can multiply a matrix by a scalar value or, alternatively, multiply two matrices using the MULTIPLY MATRIX4 statement whose format is shown in FIG-49.35.

FIG-49.35
The MULTIPLE MATRIX4 Statement



In the diagram:

- destmatno*

is an integer value giving the ID of the matrix which is to contain the result of the calculation.

Option 1:

- matno1*

is an integer value giving the ID of the first matrix involved in the calculation.
- matno2*

is an integer value giving the ID of the second matrix involved in the calculation.

The contents of *matno1* will be multiplied by the contents of *matno2* and the result stored in *destmatno*.

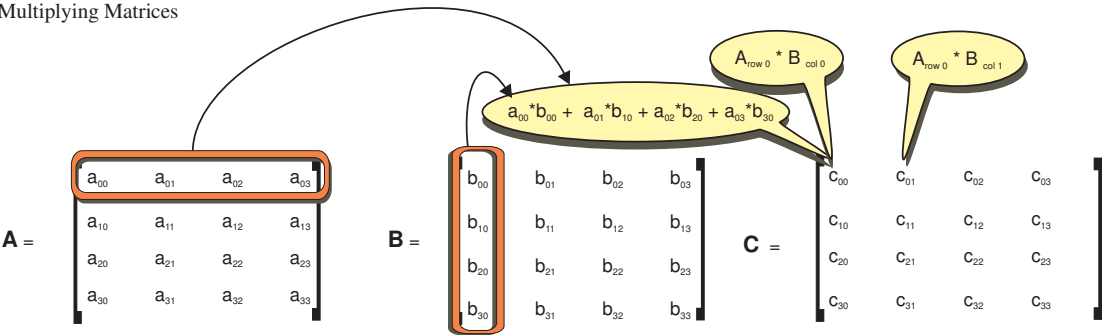
Option 2:

- multiplier*

is a real number.
- Every element in *destmatno* will be multiplied by *multiplier*.

FIG-49.36
Multiplying Matrices

Although multiplication by a scalar value is quite straight-forward, multiplying two matrices is much more complex. FIG-49.36 shows the principle involved in the operation.



Activity 49.5

What row and column would be used to calculate the value at position c21?

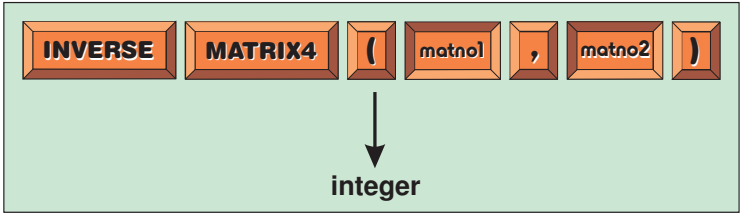
The INVERSE MATRIX4 Statement

The inverse of a scalar value is simple enough to derive; just divide it into 1. For example, the inverse of 2 is 1/2. Another property of an inverse is that when multiplied by the original value, we always get a result of 1 (2 * 1/2 = 1).

Calculating the inverse of a given matrix is more complex, but when that inverse is multiplied by the original matrix, the result is the identity matrix. Just to make life a little more complicated, not every matrix has an inverse!

We can attempt to calculate the inverse of a specified matrix using the INVERSE MATRIX4 statement whose format is shown in FIG-49.37.

FIG-49.37
The INVERSE
MATRIX4 Statement



In the diagram:

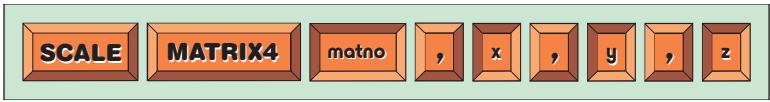
- matno1*
- is an integer value specifying the ID of the matrix in which the result is to be stored.
- matno2*
- is an integer value specifying the ID of the matrix whose contents are used in calculating the inverse matrix.

The statement returns 1 if an inverse matrix is successfully calculated; otherwise zero is returned.

The SCALE MATRIX4 Statement

Remembering that the contents of a matrix will be a triangle from a 3D object, it is possible to expand that triangle (increase its width, height and depth), using the SCALE MATRIX4 statement which has the format shown in FIG-49.38.

FIG-49.38
The SCALE MATRIX4
Statement



In the diagram:

- matno*
- is an integer value specifying the ID of the matrix to be scaled.
- x,y,z*
- are real values specifying the scaling factor for width (x), height (y) and depth (z).

For example, if matrix 1 held the coordinates of a triangle, we could double its size in all dimensions using the line:

The TRANSLATE MATRIX4 Statement

Rather than scale a triangle, we can move it using the TRANSLATE MATRIX4 statement which has the format shown in FIG-49.39.

FIG-49.39

The TRANSLATE
MATRIX4 Statement



In the diagram:

matno is an integer value specifying the ID of the matrix to be translated.

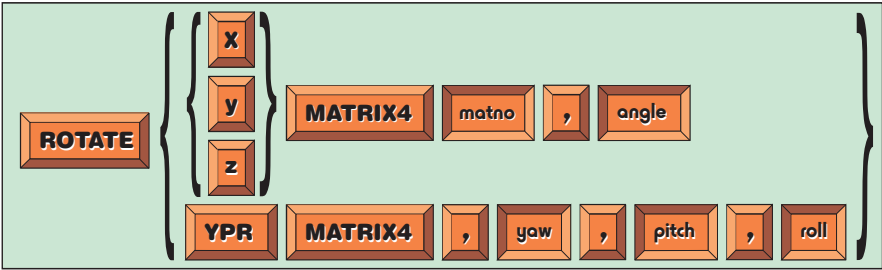
x,y,z are real values specifying the distance the coordinates are to be shifted in each dimension.

The ROTATE MATRIX4 Statement

The final effect we can apply to the triangle's coordinates is rotation. We can rotate the vertices recorded in the matrix about any axis using the ROTATE MATRIX4 statement which has the format shown in FIG-49.40.

FIG-49.40

The ROTATE
MATRIX4 Statement



In the diagram:

Option 1:

X,Y,Z Use the option corresponding to the axis about which the rotation is required.

matno is an integer value specifying the ID of the matrix to be rotated.

angle is a real value specifying the angle of rotation.

Option 2:

YPR Use this option to rotate about more than one axis.

yaw, pitch, roll are the angles of rotation about the y, x and z axes respectively (note the order!).

The TRANSPOSE MATRIX4 Statement

A transposed matrix is one in which the values making up each row are switched to make up the columns instead. A matrix **A** and its transform **A^T**, are shown below:

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

$$\mathbf{A}^T = \begin{bmatrix} 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \\ 4 & 8 & 12 & 16 \end{bmatrix}$$

To transform a matrix we use the statement TRANSFORM MATRIX4 whose format is shown in FIG-49.41.

FIG-49.41

The TRANSPOSE
MATRIX4 Statement



In the diagram:

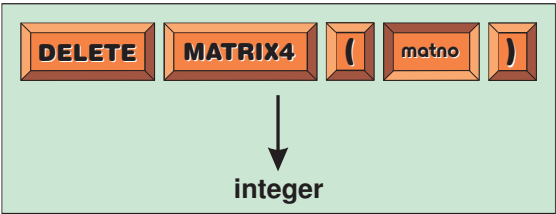
- destmatno*
- is an integer value giving the ID of the matrix which is to contain the result of the calculation
- matno*
- is an integer value specifying the ID of the matrix containing the original version of the matrix data.

The DELETE MATRIX4 Statement

A matrix can be deleted from RAM using the DELETE MATRIX4 statement whose format is given in FIG-49.42.

FIG-49.42

The DELETE MATRIX4
Statement



In the diagram:

- matno*
- is an integer value specifying the ID of the matrix to be deleted.

The statement returns 1 if the matrix is successfully deleted, otherwise zero is returned.

As stated already, the sole purpose of the matrix data structure is to be passed to a shader to modify the overall appearance of the 3D objects displayed on the screen.

Summary

- A mathematical matrix is a grid of values.
- DarkBASIC Pro has statements for creating 4 by 4 matrices only.

- The identity matrix has a diagonal row of 1's from top-left to bottom-right.
- The matrices in DarkBASIC Pro are intended to be used as parameters for shaders.
- Use MAKE MATRIX4 to create a matrix object.
- Use SET IDENTITY MATRIX4 to set the contents of a matrix to the identity matrix.
- Use IS IDENTITY MATRIX4 to determine if a matrix is an identity matrix.
- Use PROJECTION MATRIX4, VIEW MATRIX4 or WORLD MATRIX4 to assign a set of transformed coordinates from a polygon to a matrix.
- Use COPY MATRIX4 to copy the contents of one matrix to another.
- Use IS EQUAL MATRIX4 to check if two matrices have identical contents.
- Use ADD MATRIX4 to add two matrices and store the result in a third matrix.
- Use SUBTRACT MATRIX4 to subtract one matrix from another and store the result in a third matrix.
- Use DIVIDE MATRIX4 to divide a matrix by a specified value.
- Use MULTIPLY MATRIX4 to multiply two matrices and store the result in a third matrix or to multiply a matrix by a specified value.
- Use INVERSE MATRIX4 to create the inverse of an existing matrix.
- Use SCALE MATRIX4 to scale the coordinate values held within a matrix.
- Use TRANSLATE MATRIX4 to add to the coordinate values within a matrix.
- Use ROTATE MATRIX4 to rotate the coordinate values within a matrix.
- Use TRANSPOSE MATRIX4 to switch the rows and columns within a matrix.
- Use DELETE MATRIX4 to delete a matrix from RAM.

Solutions

Activity 49.1

```
GLOBAL v1#,v2#,v3#
SetUpScreen()
Read3Values()
REM *** Set up Vector ***
r = MAKE VECTOR3(1)
SET VECTOR3 1,v1#,v2#,v3#
DO
  IF UPKEY()
    ModifyVector(1,10)
  ENDIF
  IF DOWNKEY()
    ModifyVector(1,-10)
  ENDIF
  DisplayVector(1)
  WAIT 200
LOOP
WAIT KEY
END

FUNCTION SetUpScreen()
  SET DISPLAY MODE 1280,1024,32
ENDFUNCTION

FUNCTION Read3Values()
  INPUT "Enter first value : ", v1#
  INPUT "Enter second value : ", v2#
  INPUT "Enter third value : ", v3#
ENDFUNCTION

FUNCTION DisplayVector(vector)
  x# = X VECTOR3(vector)
  y# = Y VECTOR3(vector)
  z# = Z VECTOR3(vector)
  PRINT "VECTOR ",vector," : ", x#,
  " ",y#, " ",z#
ENDFUNCTION

FUNCTION ModifyVector(vector, perc#)
  multiplier# = 1 + perc#/100
  MULTIPLY VECTOR3 vector, multiplier#
ENDFUNCTION
```

Activity 49.2

The main section of the program should be changed to:

```
GLOBAL v1#,v2#,v3#
SetUpScreen()
Read3Values()
REM *** Set up Vector ***
r = MAKE VECTOR3(1)
SET VECTOR3 1,v1#,v2#,v3#
REM *** Create cube ***
MAKE OBJECT CUBE 1,10
POINT OBJECT 1, X VECTOR3(1),
  Y VECTOR3(1), Z VECTOR3(1)
DO
  IF UPKEY()
    ModifyVector(1,10)
  ENDIF
  IF DOWNKEY()
    ModifyVector(1,-10)
  ENDIF
  speed# = LENGTH VECTOR3(1)
  MOVE OBJECT 1,speed#
  SET CURSOR 100,100
  PRINT "Speed : ",speed#
LOOP
REM *** End program ***
WAIT KEY
END
```

Activity 49.3

No solution required.

Activity 49.4

[30, 9, -12, 1]

[6, 1.5, -9]

Activity 49.5

$C_{21} = A_{\text{row } 2} * B_{\text{col } 1}$

Checking Your Video Card Capabilities

Pixel Shader Statements

Using FX Files

Vertex Shader Statements

Shaders and FX Files

Introduction

Your graphics card should support pixel shader Version 2 or above.

Although DarkBASIC Pro allows us to create some spectacular effects with just a single command, more complex manipulation of an object's texture, lighting or shape needs to be done using a **shader**. A shader contains a set of instructions on how an object is to be manipulated. By using shader files, we can create photo-realistic effects that cannot be achieved by any other method. However, shaders can require a large amount of processing power and are dependent upon compliant graphics cards.

There are three stages required to use a shader:

- Create the shader
- Load the file into your program
- Apply the shader to the object

Shader files are written in a C-like language. Direct X uses the language HLSL (High-Level Shader Language) to create shaders, but other graphic-card-specific languages are also used. The compiler for HLSL can be downloaded from the Microsoft website. To use it you'll also need Visual Studio C++.NET.

Creation of a shader is beyond the scope of this text since it involves learning a whole new language, but DarkBASIC Pro has commands to load and apply these files.

There are two main types of shader files:

- vertex shaders
- pixel shaders

Several shaders can also be placed in a special effects file (FX file) so that their effects can be combined.

Vertex Shader

Graphical data sent to your graphics card goes through a great deal of processing before the final image appears on the screen. One of the earliest stages is the vertex shader which is a set of commands that manipulates the vertices of the the polygons that make up a 3D shape. By manipulating these vertices, a 3D object can be transformed, deformed or morphed. Lighting and texturing can also be altered within a vertex shader.

Pixel Shader

Once the 3D objects which make up the visible part of your world have been remapped onto a 2D plane for display on your monitor, a pixel shader can modify the colour of each individual pixel on the screen. This allows further lighting and texturing effects to be applied as well as other options such as bump mapping. As

you can imagine this requires a great deal of processing which has to be repeated every time the screen is refreshed.

FX Files

An FX file is a Microsoft DirectX effects file and it not only contains both vertex and pixel shaders, but also can define constants, parameters and texture files used by those shaders.

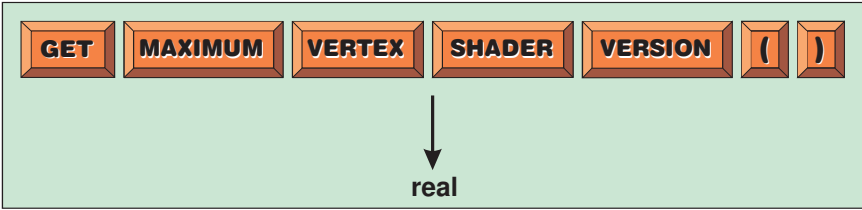
Graphics Card Check Statements

The GET MAXIMUM VERTEX SHADER VERSION Statement

One of the most important things we need to discover is exactly which version of the pixel shader and vertex shader your graphics card supports. The first of these can be determined using the GET MAXIMUM VERTEX SHADER VERSION statement which has the format shown in FIG-50.1.

FIG-50.1

The GET MAXIMUM VERTEX SHADER VERSION Statement



The statement returns a real number giving the version number of the vertex shader used by your graphics card. We can display that version number using the line:

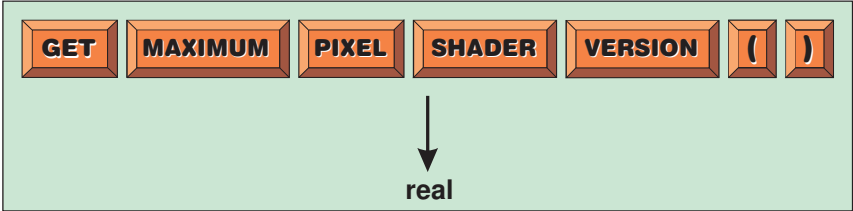
```
PRINT GET MAXIMUM VERTEX SHADER VERSION()
```

The GET MAXIMUM PIXEL SHADER VERSION Statement

The current pixel shader version can be determined using the GET MAXIMUM PIXEL SHADER VERSION statement which has the format shown in FIG-50.2.

FIG-50.2

The GET MAXIMUM PIXEL SHADER VERSION Statement



The statement returns a real number giving the version number of the pixel shader used by your graphics card. To display the version number use the line:

```
PRINT GET MAXIMUM PIXEL SHADER VERSION()
```

Activity 50.1

Write a short program (*act5001.dbpro*) to display the vertex and pixel shaders' version numbers available on your graphics card.

If you do not have version 2 or above for both shaders, then there may be problems trying to execute an FX file.

FX Statements

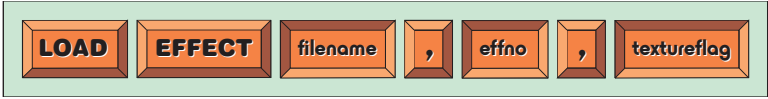
The easiest way to get started with shaders is to make use of FX files which already contain all the necessary details to execute vertex and pixel shaders.

The LOAD EFFECT Statement

To load an FX file, we use the LOAD EFFECT statement which has the format shown in FIG-50.3.

FIG-50.3

The LOAD EFFECT Statement



In the diagram:

<i>filename</i>	is a string giving the name of the FX file to be loaded. This must be a DirectX compatible shader which would normally have the file extension <i>.fx</i> .
<i>effno</i>	is an integer value giving the ID to be assigned to the effect once loaded from the file.
<i>textureflag</i>	0 or 1. If set to zero, the object to which the shader is to be applied continues to use its own textures (as specified earlier in the program). If set to 1, the object will be textured using images detailed in the shader.

For example, we could load the effects file *bubble.fx* and use the textures it defines with the line:

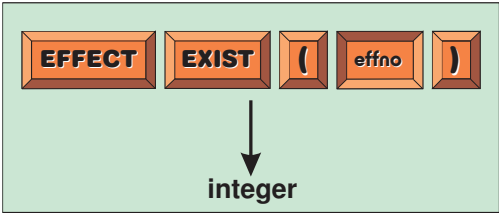
```
LOAD EFFECT "bubble.fx", 1, 1
```

The EFFECT EXIST Statement

There can be problems loading an effects file. Other than the obvious *file not found*, the contents of a file may be invalid (like any program). We can check that the effects file has been successfully loaded using the EFFECT EXIST statement which has the format shown in FIG-50.4.

FIG-50.4

The EFFECT EXIST Statement



In the diagram:

<i>effno</i>	is an integer value giving the ID of the effects object to be checked.
--------------	--

The statement returns 1 if a valid effects object of the ID specified has been created; otherwise zero is returned.

The PERFORM CHECKLIST FOR EFFECT ERRORS Statement

If an error does occur when loading an effects file, a detailed list of the problems encountered can be produced using the PERFORM CHECKLIST FOR EFFECT ERRORS statement which has the format shown in FIG-50.5.

FIG-50.5

The PERFORM
CHECKLIST FOR
EFFECT ERRORS
Statement



After this statement has been executed, we need to use the standard checklist statements to access the contents of the list.

A typical check for a successful load would be:

```
LOAD EFFECT "bubble.fx",1,1
IF EFFECT EXIST(1) = 0
    PERFORM CHECKLIST FOR EFFECT ERRORS
    FOR c = 1 TO CHECKLIST QUANTITY()
        PRINT CHECKLIST STRING$(c)
    NEXT c
    WAIT KEY
END
ENDIF
```

The SET OBJECT EFFECT Statement

With the effect object created, it can then be applied to an object using the SET OBJECT EFFECT statement which takes the format shown in FIG-50.6.

FIG-50.6

The SET OBJECT
EFFECT Statement



In the diagram:

objno

is an integer value giving the ID of the object to which the effect is to be applied.

effno

is an integer value giving the ID of the effect.

The program in LISTING-50.1 demonstrates the use of an effects file applied to a sphere.

LISTING-50.1

Demonstrating an Effect
File

```
SetUpScreen()

REM *** Create texture object ***
MAKE OBJECT SPHERE 1, 10
LOAD IMAGE "lava1.bmp",1
TEXTURE OBJECT 1,1

REM *** Load effect ***
LOAD EFFECT "bubble.fx",1,0
IF EFFECT EXIST(1) = 0
    PERFORM CHECKLIST FOR EFFECT ERRORS
```

continued on next page

LISTING-50.1

(continued)

Demonstrating an Effect
File

```
FOR c = 1 TO CHECKLIST QUANTITY()  
    PRINT CHECKLIST STRING$(c)  
    NEXT c  
    WAIT KEY  
    END  
ENDIF  
  
REM *** Apply effect after key press ***  
WAIT KEY  
SET OBJECT EFFECT 1,1  
  
REM *** end program ***  
WAIT KEY  
END  
  
FUNCTION SetUpScreen()  
    SET DISPLAY MODE 1280,1024,32  
    AUTOCAM OFF  
    POSITION CAMERA 0,5,-50  
ENDFUNCTION
```

Activity 50.2

Type in and test the program in LISTING-50.1 (*shader01.dbpro*).

The SET EFFECT ON Statement

The tasks performed by the LOAD EFFECT and SET OBJECT EFFECT statements are combined in the SET EFFECT ON statement which has the format shown in FIG-50.7.

FIG-50.7

The SET EFFECT ON
Statement



In the diagram:

- objno* is an integer value specifying the ID of the object to which the effect is to be applied.
- filename* is a string giving the name of the file containing the effect to be applied.
- textureflag* 0 or 1. If set to zero, the object to which the shader is to be applied continues to use its own textures (as specified earlier in the program). If set to 1, the object will be textured using images detailed in the shader.

Activity 50.3

Modify your last program to use the SET EFFECT ON statement in place of LOAD EFFECT and SET OBJECT EFFECT.

Although the SET EFFECT ON statement might appear to be the easier method of applying an effect it is, in fact, more inefficient since a new copy of the effect needs to be loaded for each object to which it is applied, whereas using LOAD EFFECT

and SET OBJECT EFFECT applies the same copy of the effect to each object.

The DELETE EFFECT Statement

A loaded effect can be deleted using the DELETE EFFECT statement. Obviously, an effect should not be deleted while any objects to which it has been applied still exist. The format of the DELETE EFFECT statement is shown in FIG-50.8.

FIG-50.8

The DELETE EFFECT Statement



In the diagram:

effno

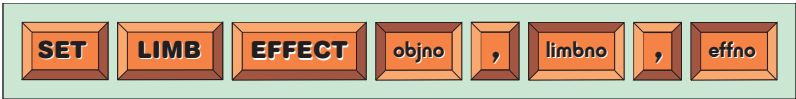
is an integer value giving the ID of the effect to be deleted.

The SET LIMB EFFECT Statement

Not only can we apply an effect to an object, but we can also apply an effect to an individual limb within a model using the SET LIMB EFFECT statement which has the format shown in FIG-50.9.

FIG-50.9

The SET LIMB EFFECT Statement



In the diagram:

objno

is an integer value giving the ID of the limb's root object.

limbno

is an integer value giving the ID of the limb.

effno

is an integer value giving the ID of the effect to be applied.

The program in LISTING-50.2 loads the ladder model we created in Chapter 35 and applies *bubble.fx* to one of its limbs.

LISTING-50.2

Applying an Effect to a Limb

```
SetUpScreen()  
REM *** Create texture object ***  
LOAD OBJECT "ladder.dbo", 1  
REM *** Apply effect after key press ***  
WAIT KEY  
LOAD EFFECT "bubble.fx", 1, 0  
SET LIMB EFFECT 1, 2, 1  
REM *** end program ***  
WAIT KEY  
END  
  
FUNCTION SetUpScreen()  
    SET DISPLAY MODE 1280, 1024, 32  
    AUTOCAM OFF  
    POSITION CAMERA 0, 5, -20  
ENDFUNCTION
```

Activity 50.4

Type in and test the program in LISTING-50.2 (*effects02.dbpro*).

The PERFORM CHECKLIST FOR EFFECT VALUES Statement

Some effects can be supplied with values in much the same way that a parameter can be passed to a function. We can discover names of any values which can be supplied to the effect using the PERFORM CHECKLIST FOR EFFECT VALUES statement which has the format shown in FIG-50.10.

FIG-50.10

The PERFORM
CHECKLIST FOR
EFFECT VALUES
Statement



We can then list the names of these values with code such as:

```
PERFORM CHECKLIST FOR EFFECT VALUES
FOR c = 1 TO CHECKLIST QUANTITY()
    PRINT CHECKLIST STRING$(c)
NEXT c
```

Activity 50.5

Write a program (*act5005.dbpro*) to display the values that may be set in the *bubble.fx* effects file.

Actually, we need more information about these named values within the FX file. Specifically, we need to know each value's type (integer, real, etc.) and if it can have its value changed from within a DarkBASIC Pro program.

By using CHECKLIST VALUE A we can discover a value's type. The statement returns a coded integer value. The code and the type it represents are shown in TABLE-50.1.

TABLE 50.1

CHECKLIST VALUE A
Return Codes

Code	Data Type
0	undefined
1	Boolean
2	integer
3	float
4	vector
5	matrix

For example, if `CHECKLIST STRING$(6) = ticks` and `CHECKLIST VALUE A(6) = 3`, then we know that variable *ticks* is a float (real) value.

Not all variables can actually be modified. Those that can, have their CHECKLIST VALUE B set to 1. And even this is not an end of the limitations; DarkBASIC Pro does not contain a statement which will allow us to assign a new value to values of an undefined type.

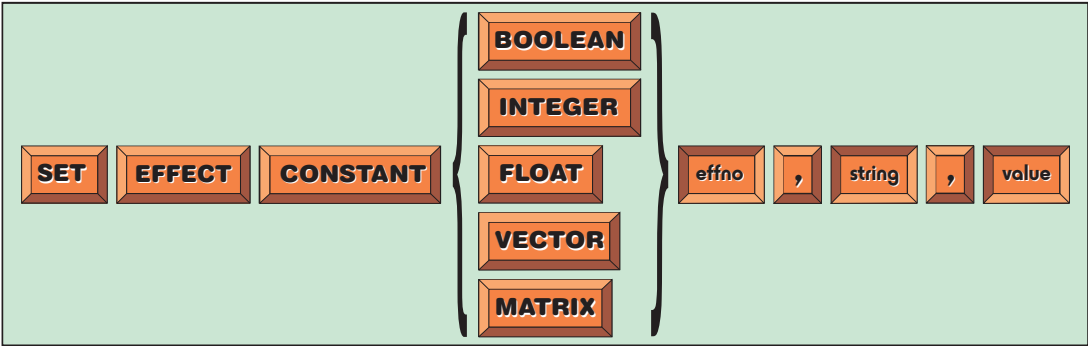
Activity 50.6

Modify your last program to display only those values which are not of an undefined type and can be modified.

The SET EFFECT CONSTANT Statement

To modify one of the parameters of an FX file, we need to use the SET EFFECT CONSTANT statement whose format is shown in FIG-50.11.

FIG-50.11 The SET EFFECT CONSTANT Statement



In the diagram:

BOOLEAN,INTEGER,FLOAT,VECTOR,MATRIX

Use the option appropriate to the type of value being assigned.

effno

is an integer value giving the ID of the effect to which a value is to be sent.

string

is a string giving the name of the parameter which is to have its value set.

value

is the value to be assigned to the parameter. This value's type must match the option chosen (Boolean, integer, float, etc.).

For example, the *bubbles.fx* file has a float parameter named *ticks*. This could have its value set to 8.5 using the lines:

```
LOAD EFFECT "bubble.fx",1,0
SET EFFECT CONSTANT FLOAT 1,"ticks",8.5
```

Actually, it's not very smart to change *ticks* since it represents the time, but the instruction itself is valid.

The SET EFFECT TECHNIQUE Statement

Some FX files contain a selection of labelled techniques (we can see these if we look at the contents of the FX file). We can choose to execute a specific technique from the file using the SET EFFECT TECHNIQUE statement which has the format shown in FIG-50.12.

FIG-50.12

The SET EFFECT TECHNIQUE Statement



In the diagram:

effno

is an integer value giving the ID of the effect in which a technique is to be specified.

name

is a string giving the name of the technique within the FX file which is to be used.

Often an FX file will contain multiple techniques with one suited to high-performance graphics cards and another for more run-of-the-mill cards. Using this command we can select the technique we wish to have executed in our program.

The SET EFFECT TRANSPOSE Statement

Some shaders expect the matrices supplied to it to be transposed, others do not. The SET EFFECT TRANSPOSE statement can be used to automatically transpose all matrices set as parameters to any shaders within an FX file. The format for the statement is given in FIG-50.13.

FIG-50.13

The SET EFFECT TRANSPOSE Statement



In the diagram:

effno

is an integer value identifying the loaded effect to which transposed matrices are to be supplied.

flag

0 or 1. Set to 1, all matrices sent to the FX file will be transposed; set to 0, matrices remain unchanged.

Vertex Shader Statements

Although using an FX file is probably the easiest way of creating a special effect, it is possible to use a vertex shader on its own without it being embedded within an FX file. This involves creating a vertex shader object loaded from a vertex shader file.

The CREATE VERTEX SHADER FROM FILE Statement

The DarkBASIC Pro examples include a single vertex shader file named *vshader.vsh*. This is a text file and can be examined using Notepad, but its unlikely to make much sense unless you have a knowledge of HLSL.

To make use of a shader, we must load the file into a vertex shader object using the CREATE VERTEX SHADER FROM FILE statement which has the format shown in FIG-50.14.

FIG-50.14 The CREATE VERTEX SHADER FROM FILE Statement



In the diagram:

vshno

is an integer value giving the ID to be assigned to the vertex shader object being created.

filename

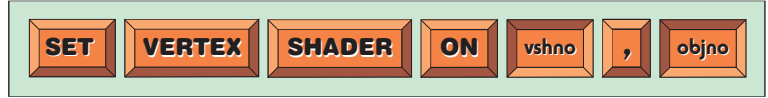
is a string giving the name of the file containing the vertex shader. Vertex shader files have a *.vsh* extension.

The SET VERTEX SHADER ON Statement

Once a vertex shader has been loaded, it still has to be applied to a specific 3D object and this is done using the SET VERTEX SHADER ON statement which takes the format shown in FIG-50.15.

FIG-50.15

The SET VERTEX SHADER ON Statement



In the diagram:

vshno

is an integer value giving the ID of the vertex shader object to be applied.

objno

is an integer value giving the ID of the 3D object to which the vertex shader is to be applied.

The SET VERTEX SHADER OFF Statement

When we no longer wish a vertex shader to be applied to an object, we can switch off the effect using the SET VERTEX SHADER OFF statement which has the format shown in FIG-50.16.

FIG-50.16

The SET VERTEX SHADER OFF Statement



In the diagram:

objno

is an integer value giving the ID of the 3D object whose vertex shader is to be switched off.

The DELETE VERTEX SHADER Statement

A vertex shader can be deleted using the DELETE VERTEX SHADER statement which has the format shown in FIG-50.17.

FIG-50.17

The DELETE VERTEX SHADER Statement



In the diagram:

vshno

is an integer value giving the ID of the vertex shader to be deleted from RAM.

A vertex shader may require data in the form of a vector or a matrix. Such information can be sent to the vertex shader using the following statements.

Other Vertex Shader Statements

There are a few other vertex shader statements designed to pass values of various types to the loaded shader objects, but without detailed knowledge of the shader's requirements these are impossible to use. The statements are:

SET VERTEX SHADER VECTOR	Passes a vector to the shader.
SET VERTEX SHADER STREAM	Sets a vertex shader stream.
SET VERTEX SHADER STREAMCOUNT	Sets the vertex shader stream count.
CONVERT OBJECT FVF	Converts a 3D object's data to a specific FVF format.

Pixel Shader Statements

A small set of statements also exist for creating pixel shader objects. These are:

CREATE PIXEL SHADER FROM FILE	Creates a pixel shader from a specified file.
DELETE PIXEL SHADER	Deletes a previously created pixel shader.
SET PIXEL SHADER ON	Activates the pixel shader.
SET PIXEL SHADER OFF	Deactivates the pixel shader.
SET PIXEL SHADER TEXTURE	Specifies an image to be used by the shader when texturing.

Summary

- Shaders are used to modify the final image that appears on the screen.
- Vertex shaders can modify the coordinates, shading and texturing of individual vertices.
- Pixel shaders can modify the colour of pixels before they appear on screen.
- Shaders are most often written in HLSL (High-Level Shader Language).
- FX files can contain both vertex and pixel shaders as well as other code.
- The shader capabilities of your system are determined by your graphics card.
- Use GET MAXIMUM VERTEX SHADER VERSION to determine the version of vertex shader available on your machine.
- Use GET MAXIMUM PIXEL SHADER VERSION to determine the version of pixel shader available on your machine.

FX Files

- Use LOAD EFFECT to load an FX file.
- Use EFFECT EXIST to check that an effect is available.

- Use **PERFORM CHECKLIST FOR EFFECT ERRORS** to create a list of errors which have been detected when attempting to load an effects file.
- Use **SET OBJECT EFFECT** to apply an effect to a specific object.
- Use **SET EFFECT ON** to load and apply an effect.
- Use **DELETE EFFECT** to delete an effect from RAM.
- Use **SET LIMB EFFECT** to apply an effect to a limb.
- Use **PERFORM CHECKLIST FOR EFFECT VALUES** to determine which values within an effect can be influenced by the program in which the effect is running.
- Use **SET EFFECT CONSTANT** to assign a value to an effect parameter.
- Use **SET EFFECT TECHNIQUE** to choose which technique within an effects file is to be executed.
- Use **SET EFFECT TRANSPOSE** to transpose a matrix being used as a parameter to an effect.

Shader Files

- Vertex and pixel shaders can exist independently of an effects file.
- Vertex and pixel shaders can be loaded and applied to specific objects.

Solutions

Activity 50.1

```
PRINT "Vertex shader version : ",
↳GET MAXIMUM VERTEX SHADER VERSION()
PRINT "Pixel shader version : ",
↳GET MAXIMUM PIXEL SHADER VERSION()
REM *** End program ***
WAIT KEY
END
```

Activity 50.2

No solution required.

Activity 50.3

```
MAKE OBJECT SPHERE 1, 10
LOAD IMAGE "lava1.bmp", 1
TEXTURE OBJECT 1, 1
REM *** Load and apply effect ***
SET EFFECT ON 1, "bubble.fx", 0
IF EFFECT EXIST(1) = 0
    PERFORM CHECKLIST FOR EFFECT ERRORS
    FOR c = 1 TO CHECKLIST QUANTITY()
        PRINT CHECKLIST STRING$(c)
    NEXT c
    WAIT KEY
END
ENDIF
REM *** end program ***
WAIT KEY
END

FUNCTION SetUpScreen()
    SET DISPLAY MODE 1280, 1024, 32
    AUTOCAM OFF
    POSITION CAMERA 0, 5, -50
ENDFUNCTION
```

Activity 50.4

No solution required.

Activity 50.5

```
SetUpScreen()
REM *** Load effect file ***
LOAD EFFECT "bubble.fx", 1, 0
REM *** Display parameters ***
PERFORM CHECKLIST FOR EFFECT VALUES 1
FOR c = 1 TO CHECKLIST QUANTITY()
    PRINT CHECKLIST STRING$(c)
NEXT c
REM *** end program ***
WAIT KEY
END

FUNCTION SetUpScreen()
    SET DISPLAY MODE 1280, 1024, 32
    AUTOCAM OFF
    POSITION CAMERA 0, 5, -20
ENDFUNCTION
```

Activity 50.6

```
SetUpScreen()
REM *** Load effect file ***
LOAD EFFECT "bubble.fx", 1, 0
REM *** Display parameters ***
PERFORM CHECKLIST FOR EFFECT VALUES 1
```

```
FOR c = 1 TO CHECKLIST QUANTITY()
    IF CHECKLIST VALUE B(c)=1
        PRINT CHECKLIST STRING$(c)
    ENDIF
NEXT c
REM *** end program ***
WAIT KEY
END

FUNCTION SetUpScreen()
    SET DISPLAY MODE 1280, 1024, 32
    AUTOCAM OFF
    POSITION CAMERA 0, 5, -20
ENDFUNCTION
```

Network Programming

Basic Networking Concepts

Choosing a Connection Method

Game Clients

Hosting a Game

Networked Games

Introduction

Back in Chapter 26 we created a two-player game, but that had both players using the same keyboard! A much better approach would be to allow two or more players to participate in the same game while sitting at different computers.

This setup gives us a lot more scope for a game; we can hide information from other players, the players can be half a world apart, and playing against another human being is always more fun than trying to best a few thousand IF statements.

Hardware Requirements

To get computers to communicate with each other, they need to be connected in some way. If only two machines are involved and they are located in the same room, then a simple cable connecting the machines' serial ports will be sufficient. Over a longer distance two machines could communicate using a phone line and modems. But when more machines are involved, then they need to be part of a **network**.

A local area network (LAN) is one where the computers are in close physical proximity to each other; perhaps in the same room, building or campus.

A wide area network (WAN) is one where the linked computers are spread over greater distances, possibly even in different countries.

In setting up a network there are choices to be made in how the machines are to communicate with each other.

In a peer-to-peer network, each machine has equal status. Information from one machine is set along a common connection and collected by the machine for which it was destined. Every machine in the network has its own unique address and the addresses of both the source computer and destination computer are sent as part of the information transmitted.

In a client/server setup, one machine acts as a server and the others as clients. The server has links to every client. While the server can communicate directly with any client, communication between clients must be routed through the server.

The client machines send requests for data to the server and the server sends back the necessary information.

The programs in this chapter have been tested on a LAN using a peer-to-peer setup. Peer-to-peer is the default setup for Microsoft Windows.

Getting Started

During the early testing stages, it is likely that we will be running our programs over machines located in the same room, since this will make testing much easier - especially if you're working alone! We need to start by making sure that all the computers that are to be used are switched on and communicating with each other.

The PERFORM CHECKLIST FOR NET CONNECTIONS Statement

The PERFORM CHECKLIST FOR NET CONNECTIONS statement looks for all possible connections to other computers that are available to the program and populates a checklist with the connections found. It will detect not only normal LAN connections but also connections between two machines using the serial port, modem connections, and connections to the Internet. The statement has the format shown in FIG-51.1.

FIG-51.1

The PERFORM
CHECKLIST FOR NET
CONNECTIONS
Statement



Once created, the contents of the checklist can be displayed using standard checklist statements.

The program in LISTING-51.1 checks for connections available on your machine.

LISTING-51.1

Listing the Connection
Options Available

```
REM *** Check for links ***
PRINT "Searching for connections..."
PERFORM CHECKLIST FOR NET CONNECTIONS

REM *** Display links found ***
PRINT CHECKLIST QUANTITY(), " Connections found"
FOR c = 1 TO CHECKLIST QUANTITY()
    PRINT c, ". ", CHECKLIST STRING$(c)
NEXT c

REM *** End program ***
WAIT KEY
END
```

Activity 51.1

Type in and test the program in LISTING-51.1 (*host01.dbpro*).

FIG-51.2 shows a screen dump of a typical output from the program above.

FIG-51.2

The Output Produced by
a Connections Check

```
Searching for connections...
4 Connections found
1. Serial Connection For DirectPlay
2. Modem Connection For DirectPlay
3. IPX Connection For DirectPlay
4. Internet TCP/IP Connection For DirectPlay
```

The IPX connection is usually installed when you have a network card (Ethernet) installed.

The TCP/IP connection will be available if you have an Internet connection.

When we choose which connection we will be using in our program, we use the corresponding number given in the listing. For example, if a machine created the list shown above, then we would use option 4 to create a TCP/IP connection between the machines. However, TCP/IP will not be listed as option 4 on every computer; perhaps the IPX option is not available on your machine; in this case, TCP/IP will be listed as option 3.

DarkBASIC Pro makes use of the DirectPlay component within DirectX to handle the networking aspects of the program.

A connection can be in a peer-to-peer or client/server setup. In either case we begin the game on one machine known as the **host**.

TCP/IP

Transmission Control Protocol/Internet Protocol is nothing more than an agreed set of standards for transmitting digital information on the Internet. The same standard is also used when transmitting data over most networks, even when there is no access to the Internet itself.

The two main parts of this standard are:

- TCP - which ensures the correct delivery of data from one machine to another. It can detect errors in transmission and cause retransmission until the data is successfully received.
- IP - which is responsible for moving data from machine to machine. The data is organised into **packets** with each packet including 4 bytes of information giving the address of the machine to which that packet is to be sent (it's a bit like an address on an envelope). Every machine on a network will have a different IP address.

Activity 51.2

In this Activity you are going to discover the IP address of your own machine.

On one of the machines you are using, create a command line window (START:RUN: Enter *cmd*).

At the command prompt, enter *ipconfig*.

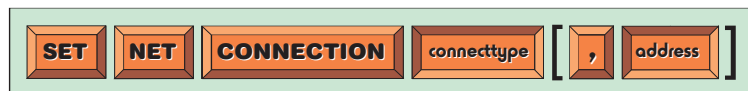
In the resulting display, under the line
Ethernet adapter Local Area Connection
there should be a line giving an IP address. It will be something similar to 192.168.0.1. Write down the IP address displayed.

The SET NET CONNECTION Statement

The computer that is to start up the game (the host or server) must specify which type of connection is being used (chosen from the list given by PERFORM CHECKLIST FOR NET CONNECTIONS). To do that, it needs to execute the SET NET CONNECTION statement which has the format shown in FIG-51.3.

FIG-51.3

The SET NET
CONNECTION
Statement



In the diagram:

connecttype

is an integer value (1 to 4) specifying which type of connection is to be used between the computers.

address

is a string, the value of which is dependent on the type of connection chosen:

(serial)	use "" (a blank string).
(modem)	use the phone number to be accessed.
(IPX)	this parameter is not used.
(TCP/IP)	the IP or URL address.

In our examples we will be using a TCP/IP connection, so we might be tempted just to write:

This assumes the machine's IP address is 192.168.0.1

```
SET NET CONNECTION 4, "192.168.0.1"
```

However, since TCP/IP will not be option 4 on every machine, a more fool-proof approach would be to create a function to find the TCP/IP option's number:

```
FUNCTION GetTCPIPNumber()  
  result = 0  
  REM *** Create list of available connections ***  
  PERFORM CHECKLIST FOR NET CONNECTIONS  
  REM *** Search for TCP/IP connection ***  
  FOR c = 1 to CHECKLIST QUANTITY()  
    IF LEFT$(CHECKLIST STRING$(c), 8) = "Internet"  
      result = c  
    ENDIF  
  NEXT c  
  REM *** IF not found, terminate program ***  
  IF result = 0  
    PRINT "No connection available. Press any key."  
    WAIT KEY  
    END  
  ENDIF  
ENDFUNCTION result
```

Now the main section of the program can create a connection using just 2 lines:

```
TCPIP = GetTCPIPNumber()  
SET NET CONNECTION TCPIP, "192.168.0.1"
```

The program in LISTING 51.2 creates a connection, displaying messages at each stage.

LISTING-51.2

Making a Connection

```
REM *** Use manual screen updating ***  
SYNC ON  
  
REM *** Get TCP/IP connection number ***  
PRINT "Checking connections available ..."  
SYNC:SYNC  
TCPIP = GetTCPIPNumber()  
PRINT "Connections found"  
SYNC  
  
REM *** Set up connection ***  
PRINT "Creating Connection "  
SYNC  
SET NET CONNECTION TCPIP, "192.168.0.1"  
PRINT "Connection created"  
SYNC  
  
REM *** End program ***  
WAIT KEY  
END
```

continued on next page

LISTING-51.2
(continued)

Making a Connection

```
FUNCTION GetTCPIPNumber()  
    result = 0  
    PERFORM CHECKLIST FOR NET CONNECTIONS  
    FOR c = 1 to CHECKLIST QUANTITY()  
        IF LEFT$(CHECKLIST STRING$(c),8) = "Internet"  
            result = c  
        ENDIF  
    NEXT c  
    IF result = 0  
        PRINT "No connection available. Press any key."  
        SYNC  
        WAIT KEY  
        END  
    ENDIF  
ENDFUNCTION result
```

When you run the program, Windows may throw up a dialog box informing you that it has blocked the program's access to the network. If this happens click on the button allowing access.

Activity 51.3

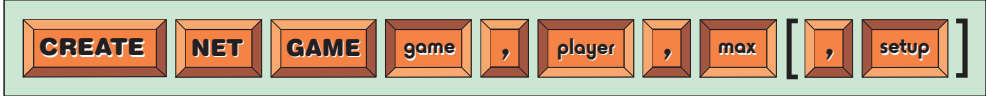
Type in and test the program in LISTING-51.2 (*host02.dbpro*).

REMEMBER TO REPLACE THE IP ADDRESS IN THE CODE WITH YOUR OWN MACHINE'S IP ADDRESS.

The CREATE NET GAME Statement

Once we've specified the connection details, we need to announce that this machine is going to host a program with which others may interact. This is done using the CREATE NET GAME statement whose format is shown in FIG-51.4.

FIG-51.4 The CREATE NET GAME Statement



In the diagram:

<i>game</i>	is a string specifying the name to be given to the program being hosted. Any name can be used.
<i>player</i>	is a string giving the name of the user at the host machine. In a game situation, this will be your character's name.
<i>max</i>	is an integer value giving the maximum number of players allowed to access the program at any one time. The maximum value is 256.
<i>setup</i>	is 1 or 2 and specifies the type of network setup: 1 peer-to-peer 2 client/server 1 is the default value.

Despite its name, this statement is used to activate any multi-terminal software - not just a game.

The *max* value determines how many machines (including the host) can be part of the link at any one time. Although a maximum of 256 is possible, a much smaller figure will probably be more practical. The maximum number of players will be determined by the speed of the game. This, in turn, is determined by the speed of

the connection between the machines, the amount of data that needs to be sent between machines, and how frequently that data must be sent. You may find for a real-time 3D game that you are limited to as little as 8 players.

In our first attempt at multi-player programming we're going to start by sending messages between two machines.

Activity 51.4

In your last program, just before the end of the main section, add the lines

```
CREATE NET GAME "Passing Messages", "John", 2, 1
PRINT "Program hosted"
SYNC
```

Writing Code for the Client Machine

So far we've created program designed to run on the machine hosting our program, now we have to create the software for the client machine.

Like the host, the client will have to detect what connection options are available and choose a TCP/IP connection, but it will be joining rather than creating a game, so it must start by discovering what games are available (that is, what games are being hosted on other machines).

The PERFORM CHECKLIST FOR NET SESSIONS Statement

We can check to see what hosted programs are available on our network using the PERFORM CHECKLIST FOR NET SESSIONS statement which has the format shown in FIG-51.5.

FIG-51.5

The PERFORM
CHECKLIST FOR NET
SESSIONS Statement



Again, we need to use the standard checklist statements to gain access to the contents of the checklist created.

Only programs hosted on other machines will be listed - programs hosted on the machine executing this statement do not appear in the list.

The program needed to detect the hosted applications available to us is given in LISTING-51.3. The program starts by creating a TCP/IP link and then displaying which net sessions are available.

LISTING-51.3

Working on a Client
Machine

```
REM *** Get TCP/IP connection number ***
TCPIP = GetTCPIPNumber()
PRINT "Connections found"

REM *** Select TCP/IP connection ***
SET NET CONNECTION TCPIP, "192.168.0.1"

REM *** List hosted programs ***
ListNetSessions()
REM *** End program ***
WAIT KEY
END
```

continued on next page

LISTING-51.3

(continued)

Working on a Client
Machine

NOTE: The SYNC
statement is not used in
this program.

```
FUNCTION GetTCPIPNumber()  
    result = 0  
    PERFORM CHECKLIST FOR NET CONNECTIONS  
    FOR c = 1 to CHECKLIST QUANTITY()  
        IF LEFT$(CHECKLIST STRING$(c),8) = "Internet"  
            result = c  
        ENDIF  
    NEXT c  
    REM *** If no connection found, stop ***  
    IF result = 0  
        PRINT "No connection available. Press any key."  
        WAIT KEY  
        END  
    ENDIF  
ENDFUNCTION result  
  
FUNCTION ListNetSessions()  
    PRINT "Sessions available : "  
    PERFORM CHECKLIST FOR NET SESSIONS  
    FOR c = 1 to CHECKLIST QUANTITY()  
        PRINT c, ". ",CHECKLIST STRING$(c)  
    NEXT c  
    PRINT "List complete"  
ENDFUNCTION
```

Notice that the SET NET CONNECTION statement gives the IP address of the host machine, not the client machine's own IP address.

Activity 51.5

On your second machine (the client machine) type in and run the program in LISTING-51.3 (*client01.dbpro*).

MAKE SURE *host02.dbpro* IS ALREADY RUNNING ON THE HOST MACHINE.

The listing produced includes two numbers at the end of each entry in the list. The entry beside *Passing Messages* should be (1/2). This tells us that the program can have up to 2 participants and that it already has 1 (the host).

The JOIN NET GAME Statement

Now that the client machine is aware of the *Passing Messages* program, it needs to start communicating with the program using the JOIN NET GAME statement which has the format shown in FIG-51.6.

FIG-51.6

The JOIN NET GAME
Statement



In the diagram:

session

is an integer value giving the program (or session) that the client machine wishes to join. This value will be taken from the list produced by the PERFORM CHECKLIST FOR NET SESSIONS statement.

player

is a string giving the name the user on the client machine wishes to be known by within the game.

For example, assuming the client machine has listed *Passing Messages* as the first item in its net sessions list, then the user at the client machine can participate in that program (using the name *Rog*) by executing the line:

```
JOIN NET GAME 1, "Rog"
```

The **PERFORM CHECKLIST FOR NET PLAYERS** Statement

We can discover the names (and other details) assigned to each participant in a session using the **PERFORM CHECKLIST FOR NET PLAYERS** statement which has the format shown in FIG-51.7.

FIG-51.7
The **PERFORM CHECKLIST FOR NET PLAYERS** Statement



The details recorded are listed in TABLE-51.1.

TABLE-51.1
Player Details

Information	Accessed by	Typical Value
User's name	CHECKLIST STRING\$()	Rog
Player's Local ID	CHECKLIST VALUE A()	1
Player's Universal ID	CHECKLIST VALUE B()	42120305
IsUser	CHECKLIST VALUE C()	1 (0 or 1)
IsHost	CHECKLIST VALUE D()	0 (0 or 1)

- User's Name

is the name the player used when joining the session.
- Player's Local ID

is the ID used to identify a user from the perspective of the machine producing this listing. For example, when a player's list is displayed on John's machine, he will have a local ID of 1 and Rog will be 2. But when the player's list is shown on Rog's machine, Rog's local ID will be 1 and John's 2.
- Player's Universal ID

is a machine-independent ID. This ID is the same irrespective of which machine lists the player's details. Every player has an unique universal ID.
- IsUser

is 1 if the player details are the user's own details; it is 0 if the details are for another player. On John's machine, the entry for John will have an *IsUser* value of 1 and the entry for Rog will be zero. On Rog's machine the entry for Rog will be 1 and for John, zero.
- IsHost

is 1 on the entry for the host machine and zero for client machines.

FIG-51.8 shows the player details produced for a session with John (host), Rog, Liz and Mary.

FIG-51.8

Player Details as Listed
at Each Machine in a
Session



Host : John

Player List

Mary	4	81270319	0	0
Liz	3	71241068	0	0
Rog	2	42120305	0	0
John	1	1	1	1



Client : Rog

Player List

Mary	4	81270319	0	0
Liz	3	71241068	0	0
Rog	1	42120305	1	0
John	2	1	0	1



Client : Liz

Player List

Mary	4	81270319	0	0
Liz	1	71241068	1	0
Rog	3	42120305	0	0
John	2	1	0	1



Client : Mary

Player List

Mary	1	81270319	1	0
Liz	4	71241068	0	0
Rog	3	42120305	0	0
John	2	1	0	1

Activity 51.6

On your client machine, modify *client01.dbpro*, so that the user joins *Passing Messages* using the name *Rog*.

Add a function, *ListPlayersDetails()*, that lists the details of all players in the current session.

Add a call to this function near the end of the main section and test the program. What numbers are assigned in the list to the two players?

MAKE SURE THE HOST IS CURRENTLY EXECUTING *host02.dbpro*.

Using a Single Machine as Both Host and Client

Are you finding it rather awkward running between two different machines? Well, we can actually make one machine act like several different machines by using a special IP address - 127.0.0.1.

Creating Window
Applications was covered
in Chapter 14.

Activity 51.7

We're going to make the programs Windows-based, since it will be easier to switch between the two programs that way.

Load *client01.dbpro*.

At the start of the program, add the lines:

```
SET WINDOW ON
SET WINDOW SIZE 500,900
SET WINDOW POSITION 501,0
SET WINDOW TITLE "CLIENT"
```

Change the line giving the IP address to use "127.0.0.1".
Save the program.

Load the program *host02.dbpro*.

At the start of the program add the lines:

```
SET WINDOW ON
SET WINDOW SIZE 500,900
SET WINDOW POSITION 0,0
SET WINDOW TITLE "HOST"
```

Change the line giving the IP address to use "127.0.0.1".
Save the program and then run it.

Switch back to DarkBASIC Pro, load *client01.dbpro* and run it.

Combining the Host/Client Requirements

Life is going to be complicated if we have to write two different programs; one to create a network connection on the host machine and another to create a connection on client machines. To get round this problem we need to write a single routine which will handle both situations. Our new function, *EstablishConnection()*, will use the following logic:

```
Get the TCP/IP connection number
Get the IP address of the host machine
Create a network connection
Get host/client choice
IF
    host
        Get user's session name
        Get maximum users
        Get session name
        Host the program
    client
        List sessions currently available
        Get session to be joined
        Get user's session name
        Join session
        List chosen session's users
ENDIF
Return host/client choice
```

This can be coded as:

```
FUNCTION EstablishConnection()
    TCPIPno = GetTCPIPNumber()
    IPaddress$ = GetIPAddress$()
    REM *** Select TCP/IP connection ***
    SET NET CONNECTION TCPIPno, IPaddress$
    option = HostOrClient()
    IF option = HOST
        REM *** Host program ***
        INPUT "Enter the name you wish to use : ", name$
        INPUT "Enter maximum users : ", max
        INPUT "Enter session name : ", session$
        CREATE NET GAME session$, name$, max, 1
    ELSE
        ListNetSessions()
        session = GetSessionToJoin()
        INPUT "Enter the name you wish to use : ", name$
        JOIN NET GAME session, name$
        ListPlayersDetails()
    ENDIF
ENDFUNCTION option
```

The code makes use of the following constants

```
#CONSTANT HOST      1
#CONSTANT CLIENT    2
```

and several routines:

```
FUNCTION GetTCPIPNumber()
    result = 0
    PERFORM CHECKLIST FOR NET CONNECTIONS
    FOR c = 1 to CHECKLIST QUANTITY()
        IF LEFT$(CHECKLIST STRING$(c), 8) = "Internet"
            result = c
        ENDIF
    NEXT c
```

```

    IF result = 0
        PRINT "No connection available. Press any key."
        WAIT KEY
    END
ENDIF
ENDFUNCTION result

FUNCTION ListNetSessions()
    PRINT "Sessions available : "
    PERFORM CHECKLIST FOR NET SESSIONS
    FOR c = 1 to CHECKLIST QUANTITY()
        PRINT c, " ",CHECKLIST STRING$(c)
    NEXT c
ENDFUNCTION

FUNCTION ListPlayersDetails()
    PRINT "Players in session "
    PERFORM CHECKLIST FOR NET PLAYERS
    FOR c = 1 to CHECKLIST QUANTITY()
        PRINT c, " Name: ",CHECKLIST STRING$(c), " Local ID : ",
            ↳CHECKLIST VALUE A(c), " Universal ID : ",
            ↳CHECKLIST VALUE B(c), " Me? : ",CHECKLIST VALUE C(c),
            ↳" Host? ",CHECKLIST VALUE D(c)
    NEXT c
ENDFUNCTION

FUNCTION HostOrClient()
    PRINT "Choose from the following options : "
    PRINT " 1 - Host a new session"
    PRINT " 2 - Join an existing session"
    INPUT "Enter option : ",option
    WHILE option < 1 OR option > 2
        PRINT "Enter 1 or 2"
        INPUT "Enter option : ",option
    ENDWHILE
ENDFUNCTION option

FUNCTION GetIPAddress$( )
    INPUT "Enter the IP address of the HOST machine
        ↳ (xxx.xxx.xxx.xxx) : ",IPaddress$
ENDFUNCTION IPaddress$

FUNCTION GetSessionToJoin()
    INPUT "Enter session you wish to join : ",session
    WHILE session < 1
        PRINT "Enter the session number from the list given"
        INPUT "Enter option : ",session
    ENDWHILE
ENDFUNCTION session

```

Activity 51.8

Type in the code given for *EstablishConnection()*, constants and called routines.

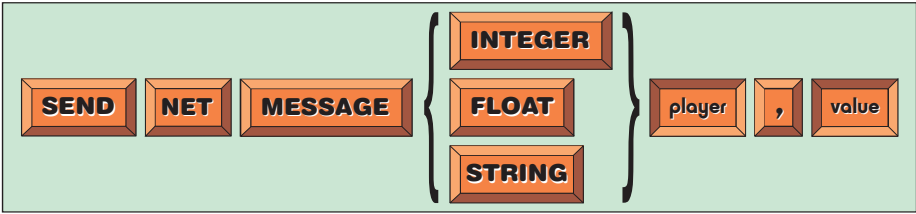
Save the file as *NetConnect.dbpro*.

We'll include this later in other network-based programs.

The SEND NET MESSAGE Statement (Version 1)

With the two users linked, we now need to exchange information between the two machines. If we wish to send standard data (integer, real, or string), then we can use version 1 of the SEND NET MESSAGE statement whose format is shown in FIG-51.9.

FIG-51.9
The SEND NET
MESSAGE
Statement



In the diagram:

INTEGER, FLOAT, STRING

Use the option appropriate for the type of value being sent.

player

is an integer value specifying the local ID of the player to receive the message.

If a value of zero is used, every member of the session (other than the person sending the message) will receive the message.

value

is the value to be sent. The value's type should match the keyword used in the statement (integer, real (float) or string).

For example, if we assume the host machine's player (John) wants to send the message "Hello there" to Rog's client machine, and that Rog has, when listed on John's machine, the local ID of 2, then the required statement that should be added to the host program is:

```
SEND NET MESSAGE 2, "Hello there, Rog"
```

On the other hand, if Rog wants to send John a message, since, from Rog's perspective, he has a local ID of 1 and John's ID is 2, a message from Rog to John would also use 2 as the first parameter:

```
SEND NET MESSAGE 2, "Hi, John"
```

The GET NET MESSAGE Statement

If one machine sends a message (using the SEND NET MESSAGE statement), the other machine needs to read that message.

Every message received by a machine is placed in a queue. To make the first message in the queue available for reading, we must start by executing the GET NET MESSAGE statement whose format is given in FIG-51.10.

FIG-51.10

The GET NET
MESSAGE Statement

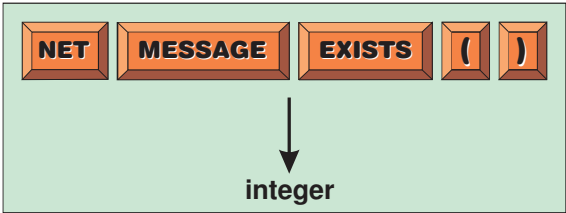


The NET MESSAGE EXISTS Statement

The big problem for any machine waiting for a message is knowing if that message has arrived. We can discover if a message has arrived using the NET MESSAGE EXISTS statement which has the format shown in FIG-51.11.

FIG-51.11

The NET MESSAGE
EXISTS Statement



The statement returns 1 if a message is available, otherwise zero is returned.

If we want a program to wait for a message arriving ,we can use the following code:

```
REPEAT
  GET NET MESSAGE
UNTIL NET MESSAGE EXISTS ()
```

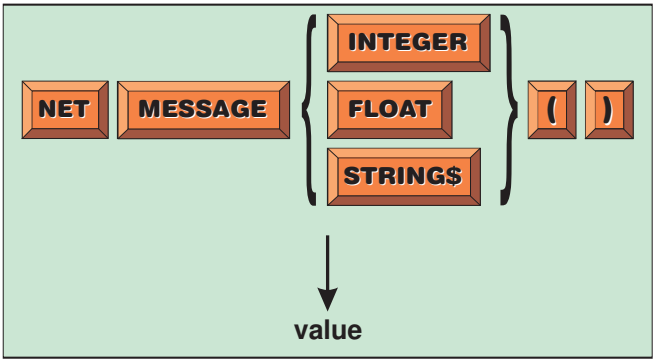
Once we have received a message, we need to read it.

The NET MESSAGE Statement (Version 1)

Reading a received message is done using the NET MESSAGE statement which has the format shown in FIG-51.12.

FIG-51.12

The NET MESSAGE
Statement



In the diagram:

INTEGER, FLOAT, STRING\$

Use the option appropriate for the type of value being read. The type must match the type of value sent by the other machine.

The type of value returned by this statement will match the type name used. That is, the statement will return an integer, real (float), or string.

The program in LISTING-51.4 sends messages between the host and client machines.

LISTING-51.4

Sending and Receiving Messages

```
#INCLUDE "NetConnect.dba"
REM *** Use a window ***
SET WINDOW ON
SET WINDOW SIZE 500,900
SET WINDOW POSITION 0,0
r = EstablishConnection()
REM *** Send and receive messages ***
DO
    REM *** Send message ***
    INPUT "Choose number of player to receive message ",playerno
    INPUT "Enter your message ", mes$
    SEND NET MESSAGE STRING playerno,mes$

    REM *** Display any received messages ***
    GET NET MESSAGE
    WHILE NET MESSAGE EXISTS ()
        PRINT NET MESSAGE STRING$ ()
        GET NET MESSAGE
    ENDWHILE
LOOP
REM *** End program ***
END
```

Activity 51.9

Type in and test the program given in LISTING-51.4 (*net04.dbpro*) on the host machine.

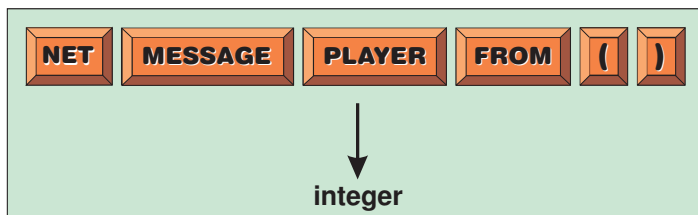
Copy the .exe version of the program onto the client machine and check that the two users can communicate with each other.

The NET MESSAGE PLAYER FROM Statement

When a message is received, we can discover who has sent it using the NET MESSAGE PLAYER FROM statement whose format is shown in FIG-51.13.

FIG-51.13

The NET MESSAGE
PLAYER FROM
Statement



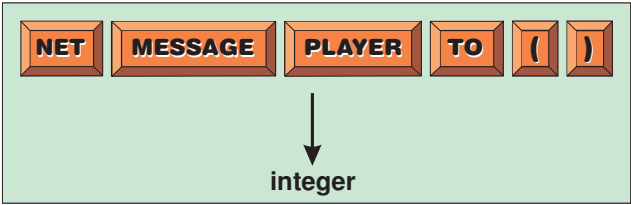
The statement returns the local ID (as seen from the receiver's perspective) of the user that sent the message which has been received.

The NET MESSAGE PLAYER TO Statement

It is also possible to discover the local ID of the message recipient using the NET MESSAGE PLAYER TO statement. This would seem a rather pointless enquiry since, if you've received a message, it was obviously meant for you and your own local ID is always 1! Even if the sender broadcasts the message to all other users (using the zero option in the SEND NET MESSAGE statement), the NET MESSAGE PLAYER TO statement still returns a value of 1. The NET MESSAGE PLAYER TO statement has the format shown in FIG-51.14.

FIG-51.14

The NET MESSAGE
PLAYER TO
Statement



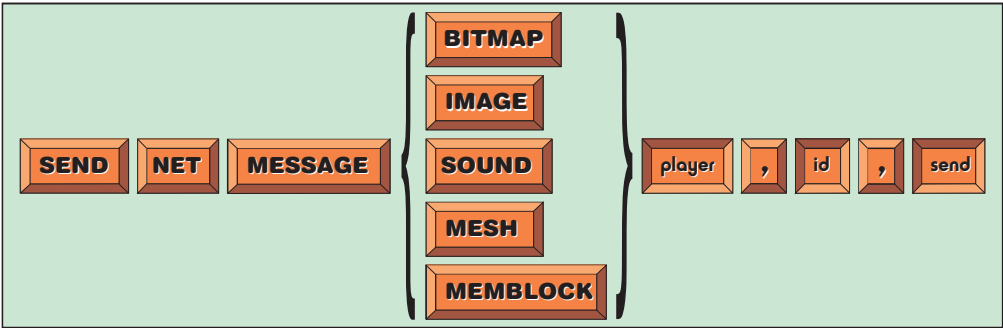
Activity 51.10

Modify your last program to display the local ID of the sender of each message.

The SEND NET MESSAGE Statement (Version 2)

As well as being able to send simple data values, it is also possible to send much more complex structures such as images, sounds, 3D objects and memory blocks. This is achieved using the second version of the SEND NET MESSAGE statement whose format is given in FIG-51.15.

FIG 51.15 The SEND NET MESSAGE Statement



In the diagram:

BITMAP, IMAGE, SOUND, MESH, MEMBLOCK

Use the option appropriate for the type of message being sent.

player

is an integer value specifying the player to receive the message. The number allocated to a player can be found using the checklist produced by the PERFORM CHECKLIST FOR NET PLAYERS statement.
If a value of zero is used, every member of the session (other than the person sending the message) will receive the message.

id

is an integer value giving the ID of the object being sent. The object must already exist.

send

0 or 1. With items which may be split into several data packets for transmission over the network, there is a chance that data will not be correctly received or may not be sent at all because of backlogs on a slow network. By setting *send* to

1 the network is forced to transmit the data and ensure it is correctly received. Setting *send* to zero means there are no guarantees of correct transmission.

For example, to transmit the details of a cube to all other machines in a session, we could use the following lines:

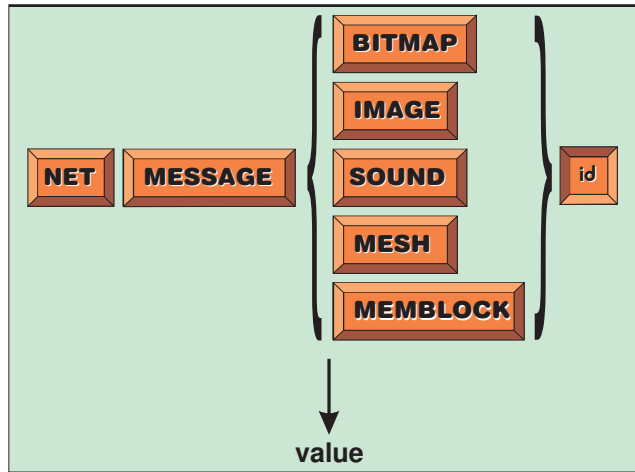
```
MAKE OBJECT CUBE 1,10
MAKE MESH FROM LIMB 1,0
SEND NET MESSAGE MESH 0,1,1
```

The NET MESSAGE Statement (Version 2)

To receive large data items we need to use a second version of the NET MESSAGE statement, the format of which is given in FIG-51.16.

FIG-51.16

The NET MESSAGE Statement



In the diagram:

BITMAP, IMAGE, SOUND, MESH, MEMBLOCK

Use the option appropriate for the type of message being received.

id

is an integer value giving the ID to be assigned to the object being received.

In the next program (LISTING-51.5) we are going to send images between the host and client.

LISTING-51.5

Sending an Image

```
#INCLUDE "NetConnect.dba"
REM *** Use a window ***
SET WINDOW ON
SET WINDOW SIZE 500,900
SET WINDOW POSITION 0,0
r = EstablishConnection()
REM *** Load images ***
LOAD IMAGE "image1.jpg",1
LOAD IMAGE "image2.bmp",2
DO
    REM *** Get recipient and image ***
    INPUT "Choose number of player to receive message ",playerno
```

continued on next page

LISTING-51.5
(continued)

Sending an Image

```
INPUT "Which image 1 or 2", imageno
REM *** Send message ***
SEND NET MESSAGE IMAGE playerno,imageno,1
REM *** Get message ***
GET NET MESSAGE
WHILE NET MESSAGE EXISTS()
    NET MESSAGE IMAGE 3
    PASTE IMAGE 3,0,0
    GET NET MESSAGE
ENDWHILE
LOOP
REM *** End program ***
END
```

Activity 51.11

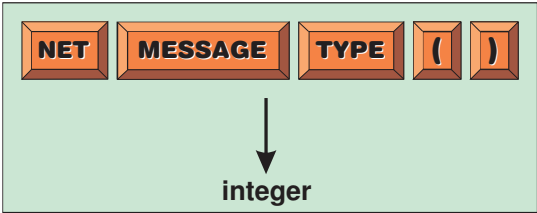
Type in and test the program in LISTING-51.5 (*net05.dbpro*).

The NET MESSAGE TYPE Statement

So far, we've written programs that know what type of data to expect (string, image, etc.), but a more flexible program needs to be able to receive and handle any type of message. Using the NET MESSAGE TYPE we can discover the type of message that has been received. This statement has the format shown in FIG-51.17.

FIG-51.17

The NET MESSAGE
TYPE Statement



This statement returns an integer value based on the type of the latest message to be received using the GET MESSAGE statement. The value returned is coded as shown in TABLE-51.2.

TABLE-51.2

The Codes Returned by
the NET MESSAGE
TYPE Statement

Code	Message Type
1	integer
2	real (float)
3	string
4	memory block
5	image
6	bitmap
7	sound
8	mesh

Using this statement, we can create a much more intelligent communications program. First we'll create a routine which can send any type of message. This function, *SendMessage()*, takes as parameters the recipient's ID, the message type (1 to 8) and the message value as a string. For objects, the message value will be the ID of the object to be sent. The function then determines which type of message is to be sent and uses the appropriate version of the SEND NET MESSAGE statement. The code for the routine is:

```
FUNCTION SendMessage(recipient, messagetype, value$)
    SELECT messagetype
    CASE 1
        value = VAL(value$)
        SEND NET MESSAGE INTEGER recipient,value
```



```

        ENDCASE
    CASE 2                                `real
        value# = VAL(value$)
        SEND NET MESSAGE FLOAT recipient, value#
    ENDCASE
    CASE 3                                `string
        SEND NET MESSAGE STRING recipient,value$
    ENDCASE
    CASE 4                                `memory block
        value = VAL(value$)
        SEND NET MESSAGE MEMBLOCK recipient,value,1
    ENDCASE
    CASE 5                                `image
        value = VAL(value$)
        SEND NET MESSAGE IMAGE recipient,value,1
    ENDCASE

CASE 6                                `bitmap
    value = VAL(value$)
    SEND NET MESSAGE BITMAP recipient,value,1
ENDCASE
CASE 7                                `sound
    value = VAL(value$)
    SEND NET MESSAGE SOUND recipient,value,1
ENDCASE
CASE 8                                `mesh
    value = VAL(value$)
    SEND NET MESSAGE MESH recipient,value,1
ENDCASE
ENDSELECT
ENDFUNCTION

```

Next we need a function to receive a message, recognise the type of message and display (or play) the message. This function, *ReceiveMessage()*, is coded as:

```

FUNCTION ReceiveMessage()
    msgtype = NET MESSAGE TYPE()
    msgfrom = NET MESSAGE PLAYER FROM()
    SELECT msgtype
        CASE 1                                `integer
            PRINT NET MESSAGE INTEGER()
        ENDCASE
        CASE 2                                `real
            PRINT NET MESSAGE FLOAT()
        ENDCASE
        CASE 3                                `string
            PRINT NET MESSAGE STRING$()
        ENDCASE
        CASE 4
            NET MESSAGE MEMBLOCK 2
        ENDCASE
        CASE 5
            NET MESSAGE IMAGE 2
            PASTE IMAGE 2,0,0
            DELETE IMAGE 2
        ENDCASE
        CASE 6
            NET MESSAGE BITMAP 2
            COPY BITMAP 2,0
            DELETE BITMAP 2
        ENDCASE
        CASE 7
            NET MESSAGE SOUND 2
            PLAY SOUND 2
            WAIT KEY
            DELETE SOUND 2
        ENDCASE
        CASE 8

```

```

        NET MESSAGE MESH 2
        MAKE OBJECT SPHERE 10,0
        ADD LIMB 10,1,2
        WAIT KEY
        DELETE OBJECT 10
        DELETE MESH 2
    ENDCASE
ENDSELECT
ENDFUNCTION

```

Activity 51.12

Add the two routines given above to *NetConnect.dba*.

In the next program (see LISTING-51.6) we'll demonstrate more flexible message passing where the users can choose the type of message that is to be transmitted.

LISTING-51.6

Passing Messages of
Various Types

```

#INCLUDE "NetConnect.dba"

REM *** Use a window ***
SET WINDOW ON
SET WINDOW SIZE 500,900
SET WINDOW POSITION 0,0
EstablishConnection()
REM *** Get recipient and message ***
PRINT "Loading..."
LOAD IMAGE "image1.jpg",1
PRINT "bottlebrush"
LOAD BITMAP "image2.jpg",1
SET CURRENT BITMAP 0
PRINT "Cactus"
LOAD SOUND "welcome.wav",1
PRINT "Sound"
REM *** Send message ***
DO
    INPUT "Choose number of player to receive message ",playerno
    INPUT "Which type of message (1..8) :", messagetype
    INPUT "Message value : ",value$
    SendMessage(playerno,messagetype,value$)
    GET NET MESSAGE
    WHILE NET MESSAGE EXISTS()
        ReceiveMessage()
        GET NET MESSAGE
    ENDWHILE
LOOP
REM *** End program ***
WAIT KEY
END

```

Activity 51.13

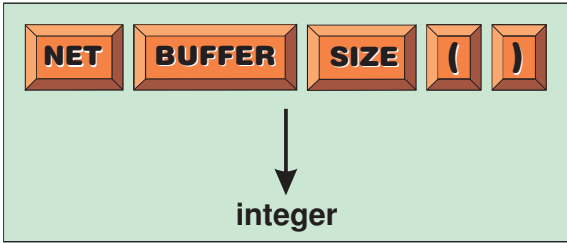
Type in and test the program in LISTING-51.6 (*net06.dba*).

The NET BUFFER SIZE Statement

If messages are being sent frequently, there's always the chance that a queue of messages begin to build up, awaiting their turn to be transmitted. Too large a build up will result in delays and maybe even the loss of messages. We can discover how many items are waiting in that queue using the NET BUFFER SIZE statement which has the format shown in FIG-51.18.

FIG-51.18

The NET BUFFER
SIZE Statement



If the statement returns zero, this means that no messages are currently awaiting transmission. So, we could check how many messages are currently in the queue using a statement such as:

```
PRINT "Messages awaiting transmission : ",NET BUFFER SIZE()
```

Session Dynamics

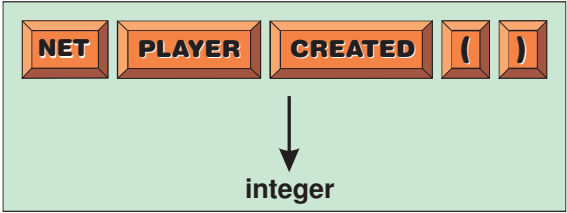
If you were writing a multi-player on-line game, you'd expect new players to arrive and existing players to leave during the game. To handle this we need statements which detect and handle such events.

The NET PLAYER CREATED Statement

We can detect if a new player has joined a session using the NET PLAYER CREATED statement which returns the local ID value of any new player joining a session. The statement has the format shown in FIG-51.19.

FIG-51.19

The NET PLAYER
CREATED Statement



If no new player has joined, the statement returns the value zero.

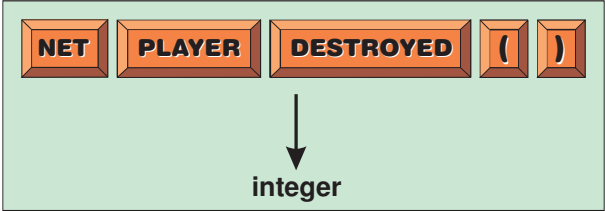
Every time a player joins a session, his details will be added as the first entry in the checklist created by the PERFORM CHECKLIST FOR NET PLAYERS statement.

The NET PLAYER DESTROYED Statement

When a player leaves a session, his local ID number can be discovered using the NET PLAYER DESTROYED statement which has the format shown in FIG-51.20.

FIG-51.20

The NET PLAYER
DESTROYED Statement



The statement returns zero if no one has left the session.

The program in LISTING-51.7 maintains a list of the players in a session, recording their local ID numbers and names. Every time a new player joins, a message and

the latest list of all players are displayed.

LISTING-51.7

Maintaining a List of
Connected Users

```
#INCLUDE "NetConnect.dba"
REM *** Structure used to hold player's details ***
TYPE PlayerDetailsType
    id AS INTEGER
    name$ AS STRING
ENDTYPE
REM *** Details of all players ***
GLOBAL DIM players(10) AS PlayerDetailsType
*** Number of players currently in session ***
GLOBAL noofentries AS INTEGER
REM *** Use a window ***
SET WINDOW ON
SET WINDOW SIZE 500,900
SET WINDOW POSITION 0,0
r = EstablishConnection()
DO
    REM *** If new player, display message and updated list ***
    IF NET PLAYER CREATED() <> 0
        PERFORM CHECKLIST FOR NET PLAYERS
        PRINT CHECKLIST STRING$(1)," has joined the session"
        MaintainPlayerList()
        DisplayPlayerList()
    ENDIF
LOOP
REM *** End program ***
WAIT KEY
END

FUNCTION MaintainPlayerList()
    PERFORM CHECKLIST FOR NET PLAYERS
    noofentries = CHECKLIST QUANTITY()
    FOR c = 1 TO noofentries
        players(c).id = CHECKLIST VALUE A(c)
        players(c).name$ = CHECKLIST STRING$(c)
    NEXT c
ENDFUNCTION

FUNCTION DisplayPlayerList()
    FOR c = 1 TO noofentries
        PRINT players(c).id," ",players(c).name$
    NEXT c
ENDFUNCTION
```

Activity 51.14

Type in the program in LISTING-51.7 (*net07.dba*).

Run the program creating four players:

John (host)
Rog (client)
Liz (client)
Mary (client)

Modify the program to display the name of any player leaving the session. The player list should also be updated and displayed when a player leaves.

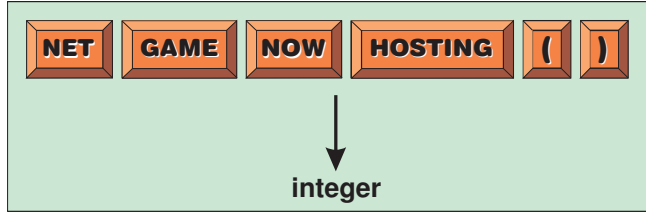
To test the new version of the program, create the same four users (as listed earlier in the question) and have them leave the session (by closing the appropriate Window) in the following order: Rog, Mary, Liz.

The NET GAME NOW HOSTING Statement

In the last Activity we deliberately retained the host player (John). But what happens if the host should leave a session? The software is ready for such a situation and simply transfers the hosting duties to one of the other members of the session. We can discover if a machine has become the host using the NET GAME NOW HOSTING statement which has the format shown in FIG-51.21.

FIG-51.21

The NET GAME NOW HOSTING Statement



This statement will return 1 if the current user has become the new host for a session; otherwise zero is returned.

Activity 51.15

Modify your last program so that the message "You are new host" is displayed by any user who becomes the new host for a session.

To test the program, create John, Rog, Liz, and Mary as before.

Close Rog, then John.

Who becomes the host?

Close Liz.

The FREE NET GAME Statement

Actually, rather than just terminate a program handling one of the players in a session, we should really close that player's link in a more organised way, freeing up any resources used by the program. To do this we can use the FREE NET GAME statement which has the format shown in FIG-51.22.

FIG-51.22

The FREE NET GAME Statement



Activity 51.16

In your last program, add the lines

```
IF ESCAPEKEY ()  
    EXIT  
ENDIF  
  
FREE NET GAME
```

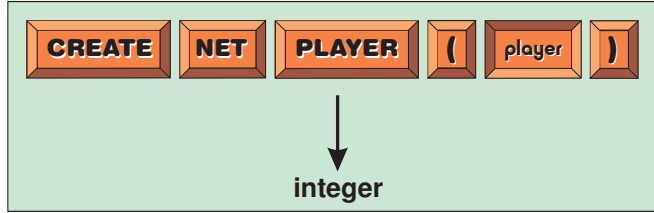
at the appropriate points so that resources are freed before a player's connection to a session is terminated.

The CREATE NET PLAYER Statement

We can add pseudo-players to a session using the CREATE NET PLAYER statement. This allows us to create a new player in a session even though that player does not represent a real user. This can be useful if a game requires a non-playing character (NPC) to be treated in the same way as real players. The format for the CREATE NET PLAYER is shown in FIG-51.23.

FIG-51.23

The CREATE NET
PLAYER Statement



In the diagram:

player

is a string giving the name to be assigned to the new session member.

The statement returns the local ID assigned to the new player.

The FREE NET PLAYER Statement

Should we want to remove a pseudo-player from the session list, we can do so using the FREE NET PLAYER statement which has the format shown in FIG-51.24.

FIG-51.24

The FREE NET PLAYER
Statement



In the diagram:

id

is an integer value giving the local ID assigned to the player.

Although it is unlikely that you'll need to use CREATE NET PLAYER or FREE NET PLAYER, we can demonstrate their use by adding the following code to the main section of our previous program:

```
REM *** Add new player (Enter key) ***
IF RETURNKEY()
  id = FindPlayerName("Avril")
  IF id = 0
    CREATE NET PLAYER ("Avril")
  ENDIF
ENDIF

REM *** Remove new player (space bar)***
IF SPACEKEY()
  id = FindPlayerName("Avril")
  IF id <> 0
    FREE NET PLAYER id
  ENDIF
ENDIF
```

The new function called by the code above is:

```

FUNCTION FindPlayerName(name$)
    result = 0
    FOR c = 1 TO noofentries
        IF players(c).name$ = name$
            result = players(c).id
            EXIT
        ENDIF
    NEXT c
ENDFUNCTION result

```

Activity 51.17

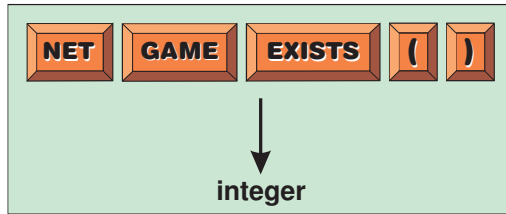
Using the code given above, modify your previous program and check that the new user can be successfully added and removed.

The NET GAME EXISTS Statement

If a session is being run on a client/server setup, closing the server will cause the whole game to be closed down. We can check that a game is still being hosted on the server using the NET GAME EXISTS statement which has the format shown in FIG-51.25.

FIG-51.25

The NET GAME
EXISTS Statement



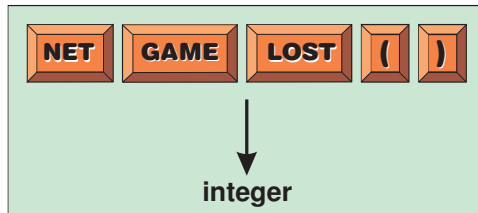
The statement returns 1 if the current game is still being hosted on a server; otherwise zero is returned.

The NET GAME LOST Statement

We can detect when a game is no longer being hosted using the NET GAME LOST statement which has the format shown in FIG-51.26.

FIG-51.26

The NET GAME LOST
Statement



The statement returns 1 if the current session no longer has a host; otherwise zero is returned.

Summary

- Networking involves communication between two or more machines.
- A Local Area Network (LAN) is a set of linked machines in close physical proximity.

- A Wide Area Network (WAN) is a set of linked machines spread over a large physical area.
- Networks can be set up in a peer-to-peer or client/server configuration.
- Use `PERFORM CHECKLIST FOR NET CONNECTIONS` to create a checklist of connections available on your system.
- Use `SET NET CONNECTION` to create a connection to a machine with a specified IP address.
- Use `CREATE NET GAME` to host a program.
- Use `PERFORM CHECKLIST FOR NET SESSIONS` to create a checklist of programs currently being hosted on other machines.
- Use `JOIN NET GAME` to join a hosted program as a client.
- Use `PERFORM CHECKLIST FOR NET PLAYERS` to create a checklist of other members of the program you have joined.
- Use `SEND NET MESSAGE` to send data to another machine in the current session.
- Use `NET MESSAGE EXISTS` to check if a message has been received.
- Use `NET MESSAGE` to read a message which has been received.
- Use `NET MESSAGE PLAYER FROM` to get the ID of the player from whom the latest message was received.
- Use `NET MESSAGE PLAYER TO` to get the ID of the player to whom the last message was directed.
- Use `NET MESSAGE TYPE` to determine what type of data is in the latest message to be received.
- Use `NET BUFFER SIZE` to determine the number of messages waiting to be transmitted.
- Use `NET PLAYER CREATED` to detect a new user joining the current session.
- Use `NET PLAYER DESTROYED` to detect a user leaving a session.
- Use `NET GAME NOW HOSTING` to discover if your machine is the host of the current session.
- Use `FREE NET GAME` to withdraw from the current session.
- Use `CREATE NET PLAYER` to create a pseudo-user in the current session.
- Use `FREE NET PLAYER` to withdraw a pseudo-user from the current session.
- Use `NET GAME EXISTS` to check that the server is still active in a client/server setup.
- Use `NET GAME LOST` to detect when a session no longer has a host.

A Networked Game

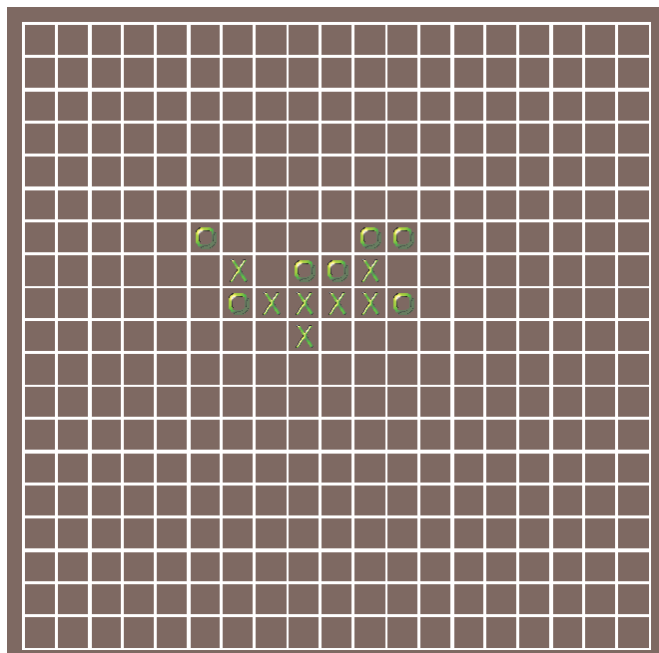
Introduction

Gomoku is a Japanese game similar to Noughts and Crosses (Tick-Tack-Toe) but instead of having to get 3 X's or O's in a row, you need to get 5.

The board is a 19 by 19 grid. On each turn a player writes an X or an O into one of the squares. The first player to create a continuous line of 5 symbols (horizontally, diagonally, or vertically) is the winner. A screen dump of the game is shown in FIG-51.27.

FIG-51.27

Gomoku



The game will be implemented using sprites. The board will be a sprite, as will each X and O symbol placed on the board.

A Non-Networked Version

We'll start by creating a non-networked version of the game and then add in the networking requirements later once we have the game itself working.

Program Data

We need to set up named constants for such things as the images and sprites used in the game. Also we need a record structure defined to store details about a player - a player ID number and the ID of the image used by that player (X image or O image). Finally, we need a set of global variables to hold the current state of the board, the position of the latest move and the ID to be used for the next sprite placed on the screen.

The code for all this is:

```

REM *****
REM ***          Constants          ***
REM *****
REM ***          Numeric            ***
#CONSTANT X          1
#CONSTANT O          2
REM ***          Images            ***
#CONSTANT ximg       1
#CONSTANT oimg       2
#CONSTANT boardimg   3
REM ***          Sprites          ***
#CONSTANT boardspr   1

REM *****
REM ***          Types              ***
REM *****
TYPE PlayerType
    value
    imgID
ENDTYPE

REM *****
REM ***          GLOBALS            ***
REM *****
GLOBAL DIM board(19,19)
GLOBAL nextsprite = 2
GLOBAL totalO
GLOBAL totalX
GLOBAL me AS PlayerType
GLOBAL you AS PlayerType
GLOBAL move

```

Game Logic

The game requires the following logic:

```

Set up details of each player
Set up screen
Set up board
REPEAT
    Get X's move
    IF X has not won THEN
        Get O's move
    ENDIF
UNTIL someone wins
End Game

```

This is coded as:

```

REM *** Main logic ***
SetUpPlayerDetails(1)
SetUpScreen()
SetUpBoard()
REPEAT
    won = -GetMove(X)
    IF NOT won
        won = GetMove(O)
    ENDIF
UNTIL won
EndGame(won)
END

```

Notice that the value returned from the first call to *GetMove()* is negated. This means *won* will be -1 if X wins and 1 if O wins. We can make use of this to choose which message should be displayed by *EndGame()*.

Activity 51.18

Create a basic program from the code given, adding level 1 test stubs for each function called.

Adding *SetUpPlayerDetails()*

The two global variables, *me* and *you*, are designed to contain details about each player. The *value* field is used to assign a value to the *board* array in the cell equivalent to where the player has placed his symbol (X or O); the *imgID* field is used to identify which image is to be used when creating a sprite on the screen representing the player's symbol.

The routine takes a value representing the parameter for *me.value* - all other assignments can be calculated from this. The code for the routine is:

```
FUNCTION SetUpPlayerDetails(r)
    REM *** Record my details ***
    me.value = r
    me.imgID = r
    REM *** Record opponents details ***
    you.value = 3-r
    you.imgID = 3-r
ENDFUNCTION
```

Adding *SetUpScreen()*

The *SetUpScreen()* function is a typical routine for setting the screen resolution and colour. Its code is:

```
FUNCTION SetUpScreen()
    SET DISPLAY MODE 1280,1024,32
    COLOR BACKDROP RGB(100,50,50)
    BACKDROP ON
ENDFUNCTION
```

Adding *SetUpBoard()*

The *SetUpBoard()* function loads the images used in the game (board, X symbol and O symbol) and displays the board using a sprite. The code is:

```
FUNCTION SetUpBoard()
    LOAD IMAGE "board2.bmp",boardimg
    LOAD IMAGE "x.bmp",ximg
    LOAD IMAGE "o.bmp",oimg
    SPRITE boardspr,200,200,boardimg
ENDFUNCTION
```

Adding *GetMove()*

The *GetMove()* function takes as a parameter an indication of which player is to add a symbol to the board. It then calls either *GetMyMove()* or *GetOpponentsMove()* as appropriate. It also checks for a winning line of 5 symbols. The code for *GetMove()* is:

```
FUNCTION GetMove(player)
    IF me.value = player
        GetMyMove()
    
```

```

ELSE
    GetOpponentsMove()
ENDIF
result = CheckForWin()
ENDFUNCTION result

```

Adding *GetMyMove()*

The *GetMyMove()* function gets the player's "move" and updates the screen by adding an X or O sprite on the board and updating the *board* array.

Reading the mouse pointer's position to determine where on the board a symbol is to be added, is actually performed by another function, *GetSquare()*, which returns the row and column of the square selected as a single integer value.

```

FUNCTION GetMyMove()
    REM *** Get a valid move ***
    REPEAT
        move = GetSquare()
        row = move /100
        col = move mod 100
        result = board(row,col)
    UNTIL result = 0
    REM *** Add sprite to screen ***
    SPRITE nextsprite,col*33+203,row*33+203,me.imgID
    REM *** Update board array ***
    board(row,col) = me.value
    REM *** next sprite ID to be used is incremented ***
    INC nextsprite
ENDFUNCTION

```

Adding *GetSquare()*

The *GetSquare()* function reads the position of the mouse pointer over the board. Each square on the board is 33 pixels by 33 pixels, so the mouse coordinates are rounded to a multiple of 33. The REPEAT..UNTIL structure ensures that the mouse has been clicked within the board area.

The coordinates are converted to row and column values and these are combined into a single value. For example, if the player clicked in row 12 column 8, this would be converted to the single value 1208. It is this single value which is returned by the function.

```

FUNCTION GetSquare()
    REM *** Get mouse position within board ***
    REPEAT
        WAIT MOUSE
        x1 = (MOUSEX()-200)/33 * 33
        y1 = (MOUSEY()-200)/33 * 33
    UNTIL InRange(x1,0,600) AND InRange(y1,0,600)
    REM *** Convert to row and column ***
    col = x1/33
    row = y1/33
    REM *** Convert row and column to a single value ***
    result = row*100+col
ENDFUNCTION result

```

Adding *InRange()*

The *InRange()* function checks to see if a value is within a given range. If it is, 1 is returned, otherwise zero is returned.

```

FUNCTION InRange(no,min,max)
  IF no >= min AND no <= max
    result = 1
  ELSE
    result = 0
  ENDIF
ENDFUNCTION result

```

Adding *GetOpponentsMove()*

The *GetOpponentsMove()* function is almost identical to *GetMyMove()*, but uses the *you* variable rather than *me*.

```

FUNCTION GetOpponentsMove()
  REM *** Get a valid move ***
  REPEAT
    move = GetSquare()
    row = move /100
    col = move mod 100
    result = board(row,col)
  UNTIL result = 0
  REM *** Add sprite ***
  SPRITE nextsprite,col*33+203,row*33+203,you.imgid
  REM *** Update board ***
  board(row,col)= you.value
  REM *** Another sprite used ***
  INC nextsprite
ENDFUNCTION

```

Activity 51.19

Add all the code given so far and any additional test stubs required.

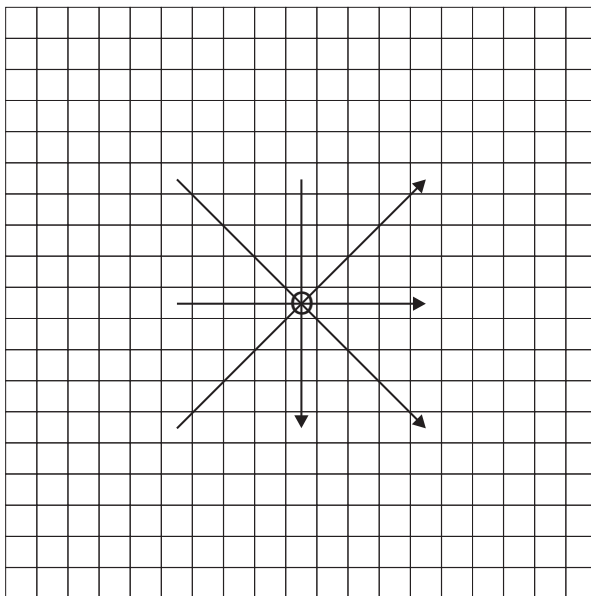
Test the program to make sure it accepts each player's move correctly.

Adding *CheckForWin()*

We need to check for a winning line of 5 symbols. This can be in a horizontal, vertical or diagonal direction (see FIG-51.28).

FIG-51.28

A line can be Created in
any One of Four
Directions



The main checking routine, *CheckForWin()* will make use of other routines to check each of these directions separately.

```
FUNCTION CheckForWin(player)
    result = 0
    result = HorizontalCheck(player)
    IF result = 0
        result = VerticalCheck(player)
    ENDIF
    IF result = 0
        result = BLTRDiagonalCheck(player)
    ENDIF
    IF result = 0
        result = TLBRDiagonalCheck(player)
    ENDIF
ENDFUNCTION result
```

Adding the Other Search Routines

We need not go into the details of how the other routines work, other than to say that each returns 1 if a row of 5 symbols is found, otherwise 0 is returned.

```
FUNCTION HorizontalCheck(player)
    REM *** Calc row and col of last move ***
    row = move / 100
    col = move mod 100
    REM *** Upper and lower limit of search area ***
    start = col - 4
    IF start < 0
        start = 0
    ENDIF
    finish = col + 4
    IF finish > 18
        finish = 18
    ENDIF
    REM *** search ***
    total = 0
    FOR col = start TO finish
        IF board(row,col)= player
            INC total
            IF total = 5
                EXIT
            ENDIF
        ELSE
            total = 0
        ENDIF
    NEXT col
    IF total = 5
        result = 1
    ELSE
        result = 0
    ENDIF
ENDFUNCTION result
```

```
FUNCTION VerticalCheck(player)
    REM *** Calc row and col of last move ***
    row = move / 100
    col = move mod 100
    REM *** Upper and lower limit of search area ***
    start = row - 4
    IF start < 0
        start = 0
    ENDIF
    finish = row + 4
```

```

    IF finish > 18
        finish = 18
    ENDIF
    REM *** search ***
    total = 0
    FOR row = start TO finish
        IF board(row,col)= player
            INC total
            IF total = 5
                EXIT
            ENDIF
        ELSE
            total = 0
        ENDIF
    NEXT row
    REM *** Determine result ***
    IF total = 5
        result = 1
    ELSE
        result = 0
    ENDIF
ENDFUNCTION result

FUNCTION BLTRDiagonalCheck(player)
    REM *** Calc row and col of last move ***
    row = move / 100
    col = move mod 100
    REM *** Upper and lower limit of search area (row)***
    srow = row
    scol = col
    FOR p = 1 TO 4
        IF srow = 18 OR scol = 0
            EXIT
        ENDIF
        INC srow
        DEC scol
    NEXT p
    frow = row
    fcol = col
    FOR p = 1 TO 4
        IF frow = 0 OR fcol = 18
            EXIT
        ENDIF
        DEC frow
        INC fcol
    NEXT p
    range = fcol-scol+1
    FOR p = 1 TO range
        IF board(srow,scol)= player
            INC total
            IF total = 5
                EXIT
            ENDIF
        ELSE
            total = 0
        ENDIF
        DEC srow
        INC scol
    NEXT p
    REM *** Determine result ***
    IF total = 5
        result = 1
    ELSE
        result = 0
    ENDIF
ENDFUNCTION result

```

```

FUNCTION TLBRDiagonalCheck(player)
  REM *** Calc row and col of last move ***
  row = move / 100
  col = move mod 100
  REM *** Upper and lower limit of search area ***
  srow = row
  scol = col
  FOR p = 1 TO 4
    IF srow = 0 OR scol = 0
      EXIT
    ENDIF
    DEC srow
    DEC scol
  NEXT p
  frow = row
  fcol = col
  FOR p = 1 TO 4
    IF frow = 18 OR fcol = 18
      EXIT
    ENDIF
    INC frow
    INC fcol
  NEXT p
  range = fcol-scol+1
  FOR p = 1 TO range
    IF board(srow,scol)= player
      INC total
      IF total = 5
        EXIT
      ENDIF
    ELSE
      total = 0
    ENDIF
    INC srow
    INC scol
  NEXT p
  REM *** Determine result ***
  IF total = 5
    result = 1
  ELSE
    result = 0
  ENDIF
ENDFUNCTION result

```

Adding *EndGame()*

A winning message is supplied by the *EndGame()* function which uses the parameter *won* to decide on which message is to be displayed.

```

FUNCTION EndGame(won)
  REM *** Find out who won ***
  IF won = -1
    winner$ = "X"
  ELSE
    winner$ = "O"
  ENDIF
  REM *** Display message for 5 seconds ***
  SET TEXT SIZE 40
  now = TIMER()
  WHILE TIMER() - now < 5000
    SET CURSOR 1000,500
    PRINT winner$, " wins!!!"
  ENDWHILE
ENDFUNCTION

```


Activity 51.20

Add the search and end game functions to your code and test the final program.

Networking the Game

Updating the main section

Perhaps, surprisingly, we don't have to make too many changes to turn our game into a networked version.

Before we start modifying our code, we need to copy the *NetConnect.dba* file we created earlier in the chapter into our game folder.

Now the main logic needs to be changed so that host and client connections are established and that the game waits for both players to be connected before continuing. The new code is:

```
r = EstablishConnection()
WaitForSecondPlayer()
SetUpPlayerDetails(r)
SetupScreen()
SetUpBoard()
REPEAT
    won = -GetMove(X)
    IF NOT won
        won = GetMove(O)
    ENDIF
UNTIL won
EndGame(won)
END
```

Adding *WaitForSecondPlayer()*

The *WaitForSecondPlayer()* function causes the program to pause until the number of players connected to the session equals 2. So the player is aware that the program hasn't stalled, a full stop is displayed every 3 seconds. The routine's code is:

```
FUNCTION WaitForSecondPlayer()
    PRINT "Waiting for second player ";
    WHILE NumberOfPlayers() <> 2
        PRINT ".";
        WAIT 3000
    ENDWHILE
ENDFUNCTION
```

Adding *NumberOfPlayers()*

The *NumberOfPlayers()* routine returns the number of entries in the checklist after executing a PERFORM CHECKLIST FOR NET PLAYERS statement. The code for *NumberOfPlayers()* is:

```
FUNCTION NumberOfPlayers()
    PERFORM CHECKLIST FOR NET PLAYERS
    result = CHECKLIST QUANTITY()
ENDFUNCTION result
```

Modifying the Call to *SetUpPlayerDetails()*

The strange thing about network programming is trying to get your head round the fact that the same code is being run on 2 or more machines and yet that code must be able to give each player an unique identity. In our game, one machine must assign its player the X symbol, while the other machine assigns the O symbol.

We get round this problem by assigning the *me* and *you* variables different values on the different machines. The value assigned depends on whether the player is using the host machine; in which case, *me.value* is set to 1, or the client machine where *me.value* is set to 2. The *you* variable is then set to the opposite of *me*. These values are then used to decide on which sprite is displayed when the player makes a move.

The modification to the call to *SetUpPlayerDetails()* is simply to use the value returned by *EstablishConnection()* as the parameter - and hence the value assigned to *me*.

Modifying *GetMyMove()*

As before *GetMyMove()* needs to get the player's move from the mouse click, display a symbol and update the *board* array. But now it must also transmit the move chosen to the other machine using the line:

```
SEND NET MESSAGE INTEGER 2,move
```

The updated version of the routine is:

```
FUNCTION GetMyMove()  
  REPEAT  
    move = GetSquare()  
    row = move /100  
    col = move mod 100  
    result = board(row,col)  
  UNTIL result = 0  
  SPRITE nextsprite,col*33+203,row*33+203,me.imgID  
  board(row,col) = me.value  
  INC nextsprite  
  REM *** Send details of move to other player ***  
  SEND NET MESSAGE INTEGER 2,move  
ENDFUNCTION
```

Modifying *GetOpponentsMove()*

The changes to the *GetOpponentsMove()* function are more fundamental since the opponent's move will be made on the other machine. The routine needs to receive details of that move over the network and then use these details to add a symbol and update *board*. This gives us the code:

```
FUNCTION GetOpponentsMove()  
  REM *** Read move from network ***  
  REPEAT  
    GET NET MESSAGE  
  UNTIL NET MESSAGE EXISTS()  
  move = NET MESSAGE INTEGER()  
  REM *** Update board ***  
  row = move/100  
  col = move mod 100  
  board(row,col)= you.value  
  REM *** Add sprite ***
```

```

        SPRITE nextsprite,col*33+203,row*33+203,you.imgid
        REM *** Another sprite used ***
        INC nextsprite
    ENDFUNCTION

```

Modifying *EndGame()*

The only change to *EndGame()* is that we should free the network connection before finishing, so we should add the line

```
FREE NET PLAYER 1
```

at the end of the routine.

Activity 51.21

Make the changes suggested and test out your game over a network.

A Complete Listing

A complete listing of the network version of the game is given in LISTING-51.8.

LISTING-51.8

Gomuku - the Networked
Version

```

#INCLUDE "NetConnect.dba"

REM *****
REM ***          Constants          ***
REM *****
REM ***          Numeric            ***
#CONSTANT X          1
#CONSTANT O          2
REM ***          Images            ***
#CONSTANT boarding   3
#CONSTANT ximg        1
#CONSTANT oimg        2
REM ***          Sprites           ***
#CONSTANT boardspr    1

REM *****
REM ***          Types              ***
REM *****
TYPE PlayerType
    value
    imgID
ENDTYPE

REM *****
REM ***          GLOBALS            ***
REM *****
GLOBAL DIM board(19,19)
GLOBAL nextsprite = 2
GLOBAL totalO
GLOBAL totalX
GLOBAL me AS PlayerType
GLOBAL you AS PlayerType
GLOBAL move

```

continued on next page

LISTING-51.8

(continued)

Gomuku - the Networked
Version

```
REM *** Main logic ***
r = EstablishConnection()
WaitForSecondPlayer()
SetUpPlayerDetails(r)
SetupScreen()
SetUpBoard()
REPEAT
    won = -GetMove(X)
    IF NOT won
        won = GetMove(O)
    ENDIF
UNTIL won
EndGame(won)
END

REM *****
REM ***          Level 1          ***
REM *****

FUNCTION WaitForSecondPlayer()
    PRINT "Waiting for second player ";
    WHILE NumberOfPlayers() <> 2
        PRINT ".";
        WAIT 3000
    ENDWHILE
ENDFUNCTION

FUNCTION SetUpPlayerDetails(r)
    me.value = r
    me.imgID = r
    you.value = 3-r
    you.imgID = 3-r
ENDFUNCTION

FUNCTION SetupScreen()
    SET DISPLAY MODE 1280,1024,32
    COLOR BACKDROP RGB(100,50,50)
    BACKDROP ON
ENDFUNCTION

FUNCTION SetUpBoard()
    LOAD IMAGE "board2.bmp",boardimg
    LOAD IMAGE "x.bmp",ximg
    LOAD IMAGE "o.bmp",oimg
    SPRITE boardspr,200,200,boardimg
ENDFUNCTION

FUNCTION GetMove(player)
    IF me.value = player
        GetMyMove()
    ELSE
        GetOpponentsMove()
    ENDIF
    result = CheckForWin(player)
ENDFUNCTION result

FUNCTION EndGame(won)
    REM *** Find out who won ***
    IF won = -1
        winner$ = "X"
    ELSE
        winner$ = "O"
    ENDIF
    REM *** Display message for 5 seconds ***
    SET TEXT SIZE 40
```

continued on next page

LISTING-51.8

(continued)

Gomuku - the Networked
Version

```
now = TIMER()
WHILE TIMER() - now < 5000
    SET CURSOR 1000,500
    PRINT winner$, " wins!!!"
ENDWHILE
FREE NET PLAYER 1
ENDFUNCTION

REM *****
REM ***          Level 2          ***
REM *****

FUNCTION NumberOfPlayers()
    PERFORM CHECKLIST FOR NET PLAYERS
    result = CHECKLIST QUANTITY()
ENDFUNCTION result

FUNCTION GetMyMove()
    REPEAT
        move = GetSquare()
        row = move /100
        col = move mod 100
        result = board(row,col)
    UNTIL result = 0
    SPRITE nextsprite, col*33+203, row*33+203, me.imgID
    board(row,col) = me.value
    INC nextsprite
    REM *** Send details of move to other player ***
    SEND NET MESSAGE INTEGER 2, move
ENDFUNCTION

FUNCTION GetOpponentsMove()
    REM *** Read move from network ***
    REPEAT
        GET NET MESSAGE
    UNTIL NET MESSAGE EXISTS()
    move = NET MESSAGE INTEGER()
    REM *** Update board ***
    row = move/100
    col = move mod 100
    board(row,col)= you.value
    REM *** Add sprite ***
    SPRITE nextsprite, col*33+203, row*33+203, you.imgid
    REM *** Another sprite used ***
    INC nextsprite
ENDFUNCTION

FUNCTION CheckForWin(player)
    result = 0
    result = HorizontalCheck(player)
    IF result = 0
        result = VerticalCheck(player)
    ENDIF
    IF result = 0
        result = BLTRDiagonalCheck(player)
    ENDIF
    IF result = 0
        result = TLBRDiagonalCheck(player)
    ENDIF
ENDFUNCTION result
```

continued on next page

LISTING-51.8

(continued)

Gomuku - the Networked
Version

```
REM *****
REM ***      Level 3      ***
REM *****

FUNCTION GetSquare()
  REM *** Get mouse position ***
  REPEAT
    WAIT MOUSE
    x1 = (MOUSEX()-200)/33 * 33
    y1 = (MOUSEY()-200)/33 * 33
  UNTIL InRange(x1,0,600) AND InRange(y1,0,600)
  col = x1/33
  row = y1/33
  result = row*100+col
ENDFUNCTION result

FUNCTION HorizontalCheck(player)
  REM *** Calc row and col of last move ***
  row = move / 100
  col = move mod 100
  REM *** Upper and lower limit of search area ***
  start = col - 4
  IF start < 0
    start = 0
  ENDIF
  finish = col + 4
  IF finish > 18
    finish = 18
  ENDIF
  REM *** search ***
  total = 0
  FOR col = start TO finish
    IF board(row,col)= player
      INC total
      IF total = 5
        EXIT
      ENDIF
    ELSE
      total = 0
    ENDIF
  NEXT col
  IF total = 5
    result = 1
  ELSE
    result = 0
  ENDIF
ENDFUNCTION result

FUNCTION VerticalCheck(player)
  REM *** Calc row and col of last move ***
  row = move / 100
  col = move mod 100
  REM *** Upper and lower limit of search area ***
  start = row - 4
  IF start < 0
    start = 0
  ENDIF
  finish = row + 4
  IF finish > 18
    finish = 18
  ENDIF
  REM *** search ***
  total = 0
  FOR row = start TO finish
    IF board(row,col)= player
      INC total
```

continued on next page

LISTING-51.8

(continued)

Gomuku - the Networked
Version

```
        IF total = 5
            EXIT
        ENDIF
    ELSE
        total = 0
    ENDIF
NEXT row
REM *** Determine result ***
IF total = 5
    result = 1
ELSE
    result = 0
ENDIF
ENDFUNCTION result

FUNCTION BLTRDiagonalCheck(player)
    REM *** Calc row and col of last move ***
    row = move / 100
    col = move mod 100
    REM *** Upper and lower limit of search area (row)***
    srow = row
    scol = col
    FOR p = 1 TO 4
        IF srow = 18 OR scol = 0
            EXIT
        ENDIF
        INC srow
        DEC scol
    NEXT p
    frow = row
    fcol = col
    FOR p = 1 TO 4
        IF frow = 0 OR fcol = 18
            EXIT
        ENDIF
        DEC frow
        INC fcol
    NEXT p
    range = fcol-scol+1
    FOR p = 1 TO range
        IF board(srow,scol)= player
            INC total
            IF total = 5
                EXIT
            ENDIF
        ELSE
            total = 0
        ENDIF
        DEC srow
        INC scol
    NEXT p
    REM *** Determine result ***
    IF total = 5
        result = 1
    ELSE
        result = 0
    ENDIF
ENDFUNCTION result

FUNCTION TLBRDiagonalCheck(player)
    REM *** Calc row and col of last move ***
    row = move / 100
    col = move mod 100
    REM *** Upper and lower limit of search area ***
    srow = row
    scol = col
```

continued on next page

LISTING-51.8

(continued)

Gomuku - the Networked
Version

```
FOR p = 1 TO 4
  IF srow = 0 OR scol = 0
    EXIT
  ENDIF
  DEC srow
  DEC scol
NEXT p
frow = row
fcol = col
FOR p = 1 TO 4
  IF frow = 18 OR fcol = 18
    EXIT
  ENDIF
  INC frow
  INC fcol
NEXT p
range = fcol-scol+1
FOR p = 1 TO range
  IF board(srow,scol)= player
    INC total
    IF total = 5
      EXIT
    ENDIF
  ELSE
    total = 0
  ENDIF
  INC srow
  INC scol
NEXT p
REM *** Determine result ***
IF total = 5
  result = 1
ELSE
  result = 0
ENDIF
ENDFUNCTION result

FUNCTION Minimum(a,b)
  IF a < b
    result = a
  ELSE
    result = b
  ENDIF
ENDFUNCTION result

REM *****
REM ***          Level 4          ***
REM *****

FUNCTION InRange(no,min,max)
  IF no >= min AND no <= max
    result = 1
  ELSE
    result = 0
  ENDIF
ENDFUNCTION result
```

Suggested Enhancements

As usual, there is plenty of scope for enhancing the program. An obvious addition would be a message indicating who's turn it is. Perhaps the number of moves made, or the time elapsed could be displayed. Or what about being able to have the program replay all the moves in a game?

Solutions

Activity 51.1

No solution required.

Activity 51.2

No solution required.

Activity 51.3

No solution required.

Activity 51.4

No solution required.

Activity 51.5

No solution required.

Activity 51.6

```
REM *** Get TCP/IP connection number ***
TCPIP = GetTCPIPNumber()
PRINT "Connections found"
REM *** Select TCP/IP connection ***
SET NET CONNECTION TCPIP,"192.168.0.1"
REM *** List hosted programs ***
ListNetSessions()
JOIN NET GAME 1,"Rog"
ListPlayersDetails()
REM *** End program ***
WAIT KEY
END

FUNCTION GetTCPIPNumber()
    result = 0
    PERFORM CHECKLIST FOR NET CONNECTIONS
    FOR c = 1 to CHECKLIST QUANTITY()
        IF LEFT$(CHECKLIST STRING$(c),8) =
            ⌘"Internet"
            result = c
        ENDIF
    NEXT c
    REM *** If no connection found, stop ***
    IF result = 0
        PRINT "No connection available.
            ⌘Press any key."
        WAIT KEY
        END
    ENDIF
ENDFUNCTION result

FUNCTION ListNetSessions()
    PRINT "Sessions available : "
    PERFORM CHECKLIST FOR NET SESSIONS
    FOR c = 1 to CHECKLIST QUANTITY()
        PRINT c," ",CHECKLIST STRING$(c)
    NEXT c
    PRINT "List complete"
ENDFUNCTION

FUNCTION ListPlayersDetails()
    PRINT "Players in session "
    PERFORM CHECKLIST FOR NET PLAYERS
    FOR c = 1 to CHECKLIST QUANTITY()
        PRINT c," ",CHECKLIST VALUE A(c),
            ⌘" ",CHECKLIST VALUE B(c),
            ⌘" ",CHECKLIST VALUE C(c),
```

```
⌘" ",CHECKLIST VALUE D(c)
NEXT c
ENDFUNCTION
```

Activity 51.7

The main section of *client01.dbpro* should be changed to:

```
SET WINDOW ON
SET WINDOW SIZE 500,900
SET WINDOW POSITION 501,0
SET WINDOW TITLE "CLIENT"
REM *** Get TCP/IP connection number ***
TCPIP = GetTCPIPNumber()
PRINT "Connections found"
REM *** Select TCP/IP connection ***
SET NET CONNECTION TCPIP,"127.0.0.1"
REM *** List hosted programs ***
ListNetSessions()
JOIN NET GAME 1,"Rog"
ListPlayersDetails()
REM *** End program ***
WAIT KEY
END
```

host02.dbpro's main section should be:

```
REM *** Use manual screen updating ***
SET WINDOW ON
SET WINDOW SIZE 500,900
SET WINDOW POSITION 0,0
SET WINDOW TITLE "Host"
SYNC ON
REM *** Get TCP/IP connection number ***
PRINT "Checking connections available ..."
SYNC:SYNC
TCPIP = GetTCPIPNumber()
PRINT "Connections found"
SYNC
REM *** Set up connection ***
PRINT "Creating Connection "
SYNC
SET NET CONNECTION TCPIP,"127.0.0.1"
PRINT "Connection created"
SYNC
CREATE NET GAME "Passing Messages","John",
⌘2,1
PRINT "Program hosted"
SYNC
REM *** End program ***
WAIT KEY
END
```

Activity 51.8

No solution required.

Activity 51.9

No solution required.

Activity 51.10

```
#INCLUDE "NetConnect.dba"
REM *** Use a window ***
SET WINDOW ON
SET WINDOW SIZE 500,900
SET WINDOW POSITION 0,0
SET WINDOW TITLE "Host"
r = EstablishConnection()
```

```

REM *** Send and receive messages ***
DO
    REM *** Send message ***
    INPUT "Choose number of player to
    ↵ receive message ",playerno
    INPUT "Enter your message : ",mes$
    SEND NET MESSAGE STRING playerno,mes$
    REM *** Display any received messages ***
    GET NET MESSAGE
    WHILE NET MESSAGE EXISTS()
        PRINT NET MESSAGE PLAYER FROM(),
        ↵ " : ",NET MESSAGE STRING$( )
        GET NET MESSAGE
    ENDWHILE
LOOP
REM *** End program ***
END

```

Activity 51.11

No solution required.

Activity 51.12

No solution required.

Activity 51.13

No solution required.

Activity 51.14

The main section should become:

```

#INCLUDE "NetConnect.dba"

TYPE PlayerDetailsType
    id AS INTEGER
    name$ AS STRING
ENDTYPE

GLOBAL DIM players(10) AS PlayerDetailsType
GLOBAL noofentries AS INTEGER
REM *** Use a window ***
SET WINDOW ON
SET WINDOW SIZE 500,900
SET WINDOW POSITION 0,0
r=EstablishConnection()
DO
    IF NET PLAYER CREATED()
        PERFORM CHECKLIST FOR NET PLAYERS
        PRINT CHECKLIST STRING$(1),
        ↵ " has joined the session"
        MaintainPlayerList()
        DisplayPlayerList()
    ENDIF
    playerleft = NET PLAYER DESTROYED()
    IF playerleft
        PRINT
        ↵ players(FindPlayerID(playerleft)).name$
        ↵ ", " has left the session"
        MaintainPlayerList()
        DisplayPlayerList()
    ENDIF
LOOP
REM *** End program ***
END

```

The new routine is:

```

FUNCTION FindPlayerID(id)
    FOR c = 1 TO noofentries

```

```

        IF players(c).id = id
            result = c
            EXIT
        ENDIF
    NEXT c
ENDFUNCTION result

```

Activity 51.15

The DO..LOOP of the program should be coded as:

```

DO
    IF NET PLAYER CREATED()
        PERFORM CHECKLIST FOR NET PLAYERS
        PRINT CHECKLIST STRING$(1),
        ↵ " has joined the session"
        MaintainPlayerList()
        DisplayPlayerList()
    ENDIF
    playerleft = NET PLAYER DESTROYED()
    IF playerleft
        PRINT
        ↵ players(FindPlayerID(playerleft)).name$
        ↵ ", " has left the session"
        MaintainPlayerList()
        DisplayPlayerList()
    ENDIF
    IF NET GAME NOW HOSTING()
        PRINT "You are new host"
    ENDIF
LOOP

```

When John leaves the session, Liz becomes the host.
When Liz leaves the session Mary becomes the host.

Activity 51.16

The last part of the main section should now be:

```

DO
    IF NET PLAYER CREATED()
        PERFORM CHECKLIST FOR NET PLAYERS
        PRINT CHECKLIST STRING$(1),
        ↵ " has joined the session"
        MaintainPlayerList()
        DisplayPlayerList()
    ENDIF
    playerleft = NET PLAYER DESTROYED()
    IF playerleft
        PRINT
        ↵ players(FindPlayerID(playerleft)).name$
        ↵ ", " has left the session"
        MaintainPlayerList()
        DisplayPlayerList()
    ENDIF
    IF NET GAME NOW HOSTING()
        PRINT "You are new host"
    ENDIF
    IF ESCAPEKEY()
        EXIT
    ENDIF
LOOP
REM *** End program ***
FREE NET GAME
END

```

Activity 51.17

The complete program is coded as:

```
#INCLUDE "NetConnect.dba"

TYPE PlayerDetailsType
  id AS INTEGER
  name$ AS STRING
ENDTYPE

GLOBAL DIM players(10) AS PlayerDetailsType
GLOBAL noofentries AS INTEGER
REM *** Use a window ***
SET WINDOW ON
SET WINDOW SIZE 500,900
SET WINDOW POSITION 0,0
r=EstablishConnection()
DO
  IF NET PLAYER CREATED()
    PERFORM CHECKLIST FOR NET PLAYERS
    PRINT CHECKLIST STRING$(1),
      " has joined the session"
    MaintainPlayerList()
    DisplayPlayerList()
  ENDIF
  playerleft = NET PLAYER DESTROYED()
  IF playerleft
    PRINT
    "players(FindPlayerID(playerleft))
    "name$, " has left the session"
    MaintainPlayerList()
    DisplayPlayerList()
  ENDIF
  IF NET GAME NOW HOSTING()
    PRINT players(1).name$,
      " now hosting session"
  ENDIF
  IF ESCAPEKEY()
    EXIT
  ENDIF
  REM *** Create new player (Enter key)***
  IF RETURNKEY()
    id = FindPlayerName("Avril")
    IF id = 0
      CREATE NET PLAYER ("Avril")
    ENDIF
  ENDIF
  REM *** Remove new player (space bar)***
  IF SPACEKEY()
    id = FindPlayerName("Avril")
    IF id <> 0
      FREE NET PLAYER id
    ENDIF
  ENDIF
LOOP
REM *** End program ***
FREE NET GAME
END

FUNCTION MaintainPlayerList()
  PERFORM CHECKLIST FOR NET PLAYERS
  noofentries = CHECKLIST QUANTITY()
  FOR c = 1 TO noofentries
    players(c).id = CHECKLIST VALUE A(c)
    players(c).name$ =
      "CHECKLIST STRING$(c)
  NEXT c
ENDFUNCTION

FUNCTION DisplayPlayerList()
  FOR c = 1 TO noofentries
    PRINT players(c).id,
      " ",players(c).name$
  NEXT c
ENDFUNCTION

FUNCTION FindPlayerID(id)
  FOR c = 1 TO noofentries
    IF players(c).id = id
      result = c
      EXIT
    ENDIF
  NEXT c
ENDFUNCTION
```

```
ENDIF
NEXT c
ENDFUNCTION result

FUNCTION FindPlayerName(name$)
  result = 0
  FOR c = 1 TO noofentries
    IF players(c).name$ = name$
      result = players(c).id
      EXIT
    ENDIF
  NEXT c
ENDFUNCTION result
```

Activity 51.18

```
REM *****
REM *** Constants ***
REM *****
REM *** Numeric ***
#CONSTANT X 1
#CONSTANT O 2
REM *** Images ***
#CONSTANT ximg 1
#CONSTANT oimg 2
#CONSTANT boarding 3
REM *** Sprites ***
#CONSTANT boardspr 1
REM *****
REM *** Types ***
REM *****
TYPE PlayerType
  value
  imgID
ENDTYPE

REM *****
REM *** GLOBALS ***
REM *****
GLOBAL DIM board(19,19)
GLOBAL nextsprite = 2
GLOBAL totalO
GLOBAL totalX
GLOBAL me AS PlayerType
GLOBAL you AS PlayerType
GLOBAL move

REM *** Main logic ***
SetUpPlayerDetails(1)
SetUpScreen()
SetUpBoard()
REPEAT
  won = -GetMove(X)
  IF NOT won
    won = GetMove(O)
  ENDIF
UNTIL won
EndGame(won)
END

REM *****
REM *** Level 1 ***
REM *****
FUNCTION SetUpPlayerDetails(r)
  PRINT "SetUpPlayerDetails"
ENDFUNCTION

FUNCTION SetUpScreen()
  PRINT "SetUpScreen"
ENDFUNCTION

FUNCTION SetUpBoard()
  PRINT "SetUpBoard"
ENDFUNCTION

FUNCTION GetMove(player)
```

```
    PRINT "GetMove"  
ENDFUNCTION 1  
  
FUNCTION EndGame(won)  
    PRINT "EndGame"  
ENDFUNCTION
```

Activity 51.19

The only test stub required is for *CheckForWin()*

Activity 51.20

No solution required.

Activity 51.21

The complete program is given in LISTING-51.8.

Using File Transfer Protocol

Checking FTP Connection Status

Creating an FTP Connection

Downloading and Uploading Files Using FTP

Retrieving File Information Using FTP

Internet File Transfers

Introduction

A frequent requirement of any Internet based software is to transfer files from a web site to your own machine or vice versa.

A common method of file transfer is File Transfer Protocol (FTP) and DarkBASIC Pro has a set of commands specifically designed to handle this type of situation.

These instructions allow us to connect to a web site, examine and download files from the site and even delete and add files.

Obviously, you need to be connected to the Internet (or have created your own Intranet) to use these statements.

The Instructions

The FTP CONNECT Statement

Step 1 in accessing the files on a web site is to establish a connection between your machine and that web site. This is done using the FTP CONNECT statement which has the format given in FIG-52.1.

FIG-52.1

The FTP CONNECT Statement



In the diagram:

<i>url</i>	is a string giving the web address of the site to be accessed.
<i>name</i>	is a string giving the user name required for accessing the site's files.
<i>password</i>	is a string giving the user's password for accessing the files. Use an empty string if no password is required.

Many FTP sites require you to register with them and hence the need for a name and password. However, it is also common to allow general access to many sites, in which case the name is often "anonymous" and there is no password.

For example, we could access the FTP site, "ftp:\\ftp.darkbasic.co.uk" using the line

```
FTP CONNECT "ftp:\\ftp.darkbasic.co.uk", "anonymous", ""
```

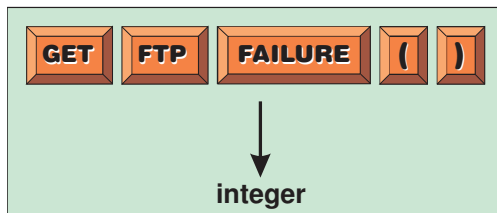
The GET FTP FAILURE Statement

There are many reasons why an attempted connection might fail; the wrong address, the site is down, your Internet connection may be faulty. So, after executing an FTP CONNECT statement, a program should check for a successful connection. This

can be done using the GET FTP FAILURE statement which has the format shown in FIG-52.2.

FIG-52.2

The GET FTP
FAILURE Statement



This statement returns 1 if an attempted FTP connection is unsuccessful; otherwise zero is returned.

For example, we could indicate failure to connect using the lines:

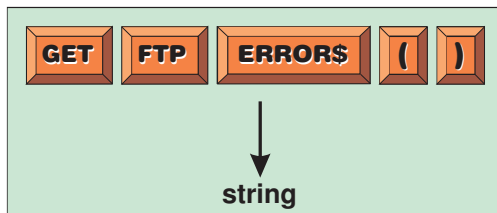
```
FTP CONNECT "ftp://ftp.darkbasic.co.uk", "anonymous", ""
IF GET FTP FAILURE()
    PRINT "Failed"
ENDIF
```

The GET FTP ERROR\$ Statement

We can find out more about a failure to connect using GET FTP ERROR\$ which will return a string detailing the reason for a failure to connect. The format for this statement is given in FIG-52.3.

FIG-52.3

The GET FTP ERROR\$
Statement



For example, rather than simply display the word *failed*, as in the last example, we could display the reason for a failure to connect using the code:

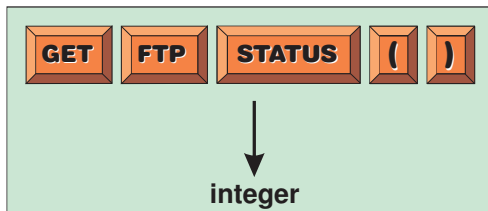
```
FTP CONNECT "ftp://ftp.darkbasic.co.uk", "anonymous", ""
IF GET FTP FAILURE()
    PRINT GET FTP ERROR$()
ENDIF
```

The GET FTP STATUS Statement

Even if a connection is successfully made, that does not mean that problems with the connection may not arise later. When working on an FTP site, you should continually check for connection problems. This can be done using the GET FTP STATUS statement which has the format shown in FIG-52.4.

FIG-52.4

The GET FTP STATUS
Statement



This statement returns 1 if the connection is still valid, and zero if the connection fails.

The FTP SET DIR Statement

Although we might have included directory details in the FTP CONNECT statement, it is also possible to move to a different directory within an FTP site after a connection has been made using the FTP SET DIR statement, which has the format shown in FIG-52.5.

FIG-52.5

The FTP SET DIR Statement



In the diagram:

path

is a string specifying the new directory to be accessed. This will become the current directory.

For example, we could change to the *members* folder within the site *ftp:\ftp.digital-skills.co.uk* using the lines:

```
FTP CONNECT "ftp:\\ftp.digital-skills.co.uk","anonymous",""  
FTP SET DIR "members"
```

It is possible to move one level up the folder hierarchy using the path *..*. So, if the current folder was *members* we could return to the original level using the line:

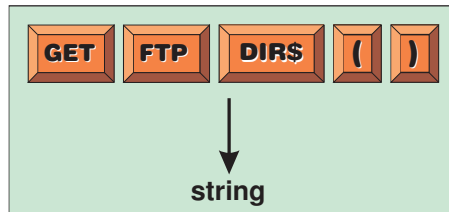
```
FTP SET DIR ".."
```

The GET FTP DIR\$ Statement

We can discover which folder is the current directory using the GET FTP DIR\$ statement which has the format shown in FIG-52.6.

FIG-52.6

The GET FTP DIR\$ Statement



This statement will return a string detailing the current folder and path. A typical statement would be:

```
currentfolder$ = GET FTP DIR$ ()
```

The FTP FIND FIRST Statement

Once within a folder, we can select a specific file in various ways. One option is to move through the list of entries within the folder until the required file is selected. We can select the first file within the current folder using the FTP FIND FIRST statement which has the format shown in FIG-52.7.

FIG-52.7

The FTP FIND FIRST Statement



For example, we could move to the first file within the *members* folder using the lines:

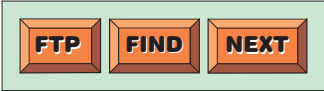

```
FTP CONNECT "ftp:\\ftp.digital-skills.co.uk", "anonymous", ""
FTP SET DIR "members"
FTP FIND FIRST
```

The FTP FIND NEXT Statement

We can select the next file within the current folder using the FTP FIND NEXT statement whose format is given in FIG-52.8.

FIG-52.8

The FTP FIND NEXT Statement

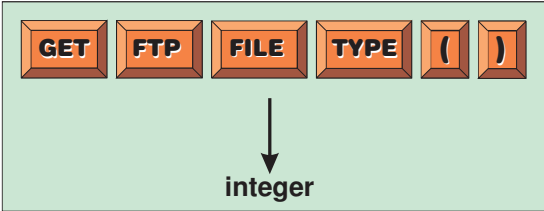


The GET FTP FILE TYPE Statement

The currently selected file's type can be determined using the GET FTP FILE TYPE statement which has the format shown in FIG-52.9.

FIG-52.9

The GET FTP FILE TYPE Statement



The statement returns one of the values shown in TABLE-52.1.

TABLE-52.1

Possible Return Values from FTP GET FILE TYPE

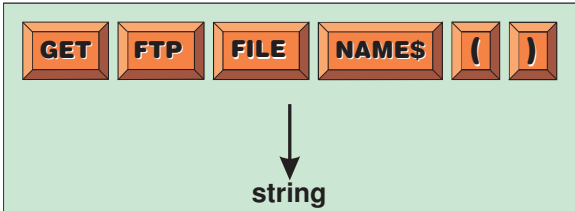
Code	File Type
-1	No more files
0	Normal file
1	Sub-folder

The GET FTP FILE NAME\$ Statement

The name of the currently selected file can be discovered using the GET FTP FILE NAME\$ statement which has the format shown in FIG-52.10.

FIG-52.10

The GET FTP FILE NAME\$ Statement

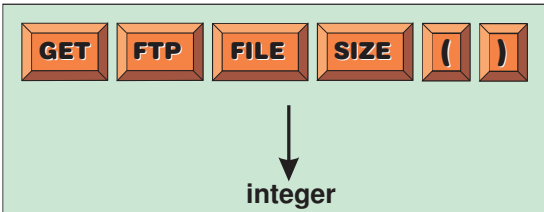


The GET FTP FILE SIZE Statement

The size of the currently selected file can be determined using the GET FTP FILE SIZE statement which has the format shown in FIG-52.11.

FIG-52.11

The GET FTP FILE SIZE Statement



The size of the file is given in bytes.

With the following code we could display the details of every file within the *members* folder:

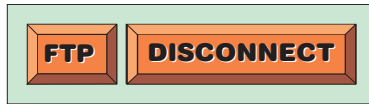
```
FTP CONNECT "ftp:\\ftp.digital-skills.co.uk","anonymous",""
FTP SET DIR "members"
FTP FIND FIRST
WHILE GET FTP FILE TYPE() <>-1
    PRINT "TYPE:";GET FTP FILE TYPE()
    PRINT "NAME:";GET FTP FILE NAME$()
    PRINT "SIZE:";GET FTP FILE SIZE()
    FTP FIND NEXT
ENDWHILE
```

The FTP DISCONNECT Statement

To terminate connection to an FTP site, we need to use the FTP DISCONNECT statement which has the format shown in FIG-52.12.

FIG-52.12

The FTP DISCONNECT
Statement



The program in LISTING-52.1 lists the details of the files set up by The Game Creators' site specifically for testing FTP access.

LISTING-52.1

Display Details of the
Files Available

```
REM *** Connect to FTP site ***
FTP CONNECT "www.darkbasic.com","",""
REM *** IF there's a problem, show message and stop ***
IF GET FTP FAILURE()
    PRINT "Failed"
    WAIT KEY
    END
ENDIF
REM *** Find the first file ***
FTP FIND FIRST
REM *** WHILE not at end of files list ***
WHILE GET FTP FILE TYPE() <>-1
    REM *** Display the file's details ***
    PRINT "TYPE: ",GET FTP FILE TYPE()
    PRINT " NAME: ",GET FTP FILE NAME$()
    PRINT " SIZE: ",GET FTP FILE SIZE()
    REM *** Move on to next file ***
    FTP FIND NEXT
ENDWHILE
REM *** Disconnect ***
FTP DISCONNECT
REM *** End program ***
WAIT KEY
END
```

Activity 52.1

Type in and test the program in LISTING-52.1 (*ftp01.dbpro*).

The FTP GET FILE Statement

What we've covered so far allows us look at file details, but if we want access to the contents of a specific file, we need to use the FTP GET FILE statement which has the format shown in FIG-52.13.

FIG-52.13

The FTP GET FILE Statement



In the diagram:

- filename* is a string giving the name of the file to be accessed.
- localname* is a string giving the file name to be used for the copy of the file made on your own hard drive.
- blocksize* is an integer value giving the size of data blocks to be transmitted. Typically, this will be 512, 1024, 2048, etc.

For example, we could download the *welcome.msg* text file from the Game Creators' site and save it on our own machine as *temp.txt* using the program in LISTING-52.2.

LISTING-52.2

Downloading a File

```
REM *** Connect to site ***
FTP CONNECT "www.darkbasic.com", "", ""
IF GET FTP FAILURE()
  PRINT "Failed"
  WAIT KEY
END
ENDIF

REM *** Down load file ***
FTP GET FILE "welcome.msg", "temp.txt"
IF GET FTP FAILURE()=0
  PRINT "File downloaded"
ELSE
  PRINT "Download failed"
ENDIF

REM *** Disconnect ***
FTP DISCONNECT

REM *** End program ***
WAIT KEY
END
```

Activity 52.2

Type in and test the program in LISTING-52.2 (*ftp02.dbpro*).

Use Notepad to read the file you have downloaded.

The FTP PROCEED Statement

Although FTP GET FILE works well on small files, larger files need to be fetched a block at a time. In that situation we use the FTP PROCEED statement to download each block of the file. This statement has the format shown in FIG-52.14.

FIG-52.14

The FTP PROCEED Statement

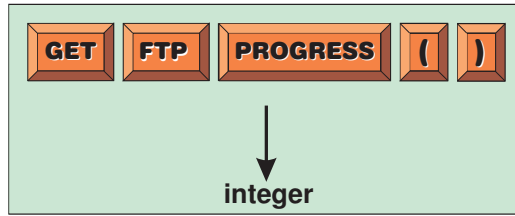


The GET FTP PROGRESS Statement

Once we start to download a file using the FTP GET FILE statement, we can check on the progress of the download using the GET FTP PROGRESS statement which has the format shown in FIG-52.15.

FIG-52.15

The GET FTP PROGRESS Statement



The statement returns the percentage of the file's contents which has been downloaded to your machine. When the download is complete, -1 is returned.

For example, let's assume that the Game Creators' area contains a file named *panorama.bmp* (the file doesn't actually exist), then we could download it using the following code:

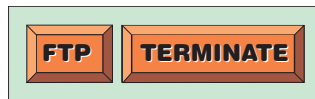
```
FTP GET FILE "panorama.bmp", "photo.bmp", 1024
IF GET FTP FAILURE () = 0
    WHILE GET FTP PROGRESS () <> -1
        CLS
        PRINT "Downloading..."
        PRINT "PROGRESS:", GET FTP PROGRESS ()
        FTP PROCEED
    ENDWHILE
ELSE
    PRINT "Download failed"
ENDIF
```

The FTP TERMINATE Statement

Should we need to halt a download before it has been completed, then we can use the FTP TERMINATE statement which has the format shown in FIG-52.16.

FIG-52.16

The FTP TERMINATE Statement



For example, we could allow the user to halt a download by pressing any key using the lines

```
IF INKEY$ () <> ""
    FTP TERMINATE
ENDIF
```

in the loop structure given on the previous page.

The FTP DELETE FILE statement

Assuming we have been assigned the appropriate rights of access, we can also delete a file on an FTP site using the FTP DELETE FILE statement which has the format shown in FIG-52.17.

FIG-52.17

The FTP DELETE FILE
Statement



In the diagram:

filename

is a string giving the name of the file to be deleted.

The FTP PUT FILE Statement

A file can also be uploaded to an FTP site (assuming we have the appropriate privileges) using the FTP PUT FILE statement which has the format shown in FIG-52.18.

FIG-52.18

The FTP PUT FILE
Statement



In the diagram:

filename

is a string giving the name of the file to be copied to the FTP site.

The code below demonstrates how a file (*myfile.dat*) might be uploaded to an FTP site:

```
FTP CONNECT "ftp://ftp.yoursite.com", "", ""
IF GET FTP FAILURE()=1
    PRINT "Could not connect"
    PRINT "Error: ";GET FTP ERROR$()
    WAIT KEY
    END
ENDIF
FTP PUT FILE "myfile.dat"
IF GET FTP FAILURE()=0
    PRINT "File uploaded"
ELSE
    PRINT "Upload failed"
ENDIF
```

Summary

- FTP stands for File Transfer Protocol.
- An FTP web site is designed to hold files and allow those with the appropriate rights to store and retrieve files from that site.
- Use FTP CONNECT to connect to an FTP site.
- Use GET FTP FAILURE to check if an attempted connection to an FTP site has failed.
- Use GET FTP ERROR\$ to access a string specifying why an attempted FTP connection has failed.
- Use GET FTP STATUS to check that an FTP connection continues to operate correctly.

- Use FTP SET DIR to change the directory being accessed on an FTP site.
- Use GET FTP DIR\$ to discover which directory within an FTP site is currently being accessed.
- Use FTP FIND FIRST to select details of the first entry in the directory being accessed.
- Use FTP FIND NEXT to select details of the next entry in the directory being accessed.
- Use GET FTP FILE TYPE to discover the type of the currently selected entry.
- Use GET FTP FILE NAME to get the name of the currently selected entry.
- Use GET FTP FILE SIZE to get the size of the currently selected entry.
- Use FTP DISCONNECT to disconnect from the FTP site being accessed.
- Use FTP GET FILE to download the contents of a file from the FTP site.
- Use FTP PROCEED to get a file's next block of data from an FTP site. This is used when large files are being downloaded.
- Use GET FTP PROGRESS to discover what percentage of a large file has been downloaded so far.
- Use FTP TERMINATE to halt the downloading of a large file.
- Use FTP DELETE FILE to delete a file from the FTP site.
- Use FTP PUT FILE to upload a file to an FTP site.

No solutions are required for this chapter.

Dynamic link libraries

Adding Help Pages to the DarkBASIC Pro IDE

Creating New DarkBASIC Pro Statements

Creating a String Table

Using Standard DLLs

Writing DLLs in C++

Creating New DBPro Statements

Introduction

If you are fluent in another programming language such as C, C++, or Delphi, it is possible to create new program language statements which can be embedded in your copy of DarkBASIC Pro. You can then use these new commands in any future DarkBASIC Pro project that you write.

Obviously, this is not the place to start teaching you another programming language, but if you already know an appropriate language, then expanding DarkBASIC Pro to meet your own specific needs is a very powerful feature.

A Dynamic Link Library (DLL)

It is possible to write a collection of functions and place them in a single library file. In fact, we've already done the same sort of thing back in Chapter 8 when we created a collection of string-handling functions.

Once created, a library can be linked to any new application being written and the function within that library can then be used by the application. One way to create such a library of functions is in a **Dynamic Link Library** or **DLL**.

In fact, all the commands in DarkBASIC Pro are held in a series of DLLs. By adding our own DLLs we can expand the statements available to us in DarkBASIC Pro.

Creating a DLL

As we stated earlier, it is possible to create a DLL library using many different programming languages, but here we are going to make use of Visual C++ .NET. Obviously, if you know C or C++ what follows will be much more meaningful, but for others, a step-by-step guide of how to add your own instructions should be useful in the future.

Starting Up Visual Studio

Visual Studio offers many options on start up, but we are going to create C++ Windows 32-bit code, so the options to be selected are those shown in FIG-53.1.

Specifically, we need to create a DLL (rather than an application), so we need to select the option shown in the dialog box that appears (see FIG-53.2).

At last we have created the project we require. By double clicking on the source code file listed in the folder-structure on the left-hand side of the screen, we can bring the file's contents into the edit window (see FIG-53.3). Notice that a few lines have been created automatically by Visual Studio.

Activity 53.1

Following the instructions given in FIG-53.1 to FIG-53.3, create a DLL named *StringFunctions*.

FIG-53.1

Choosing the Basic Options

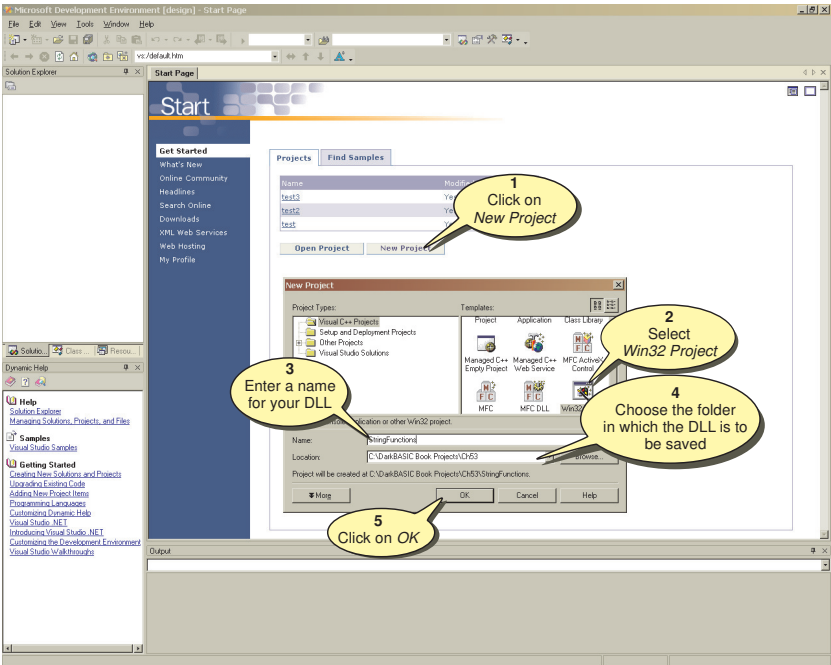


FIG-53.2

Specifying the DLL Option

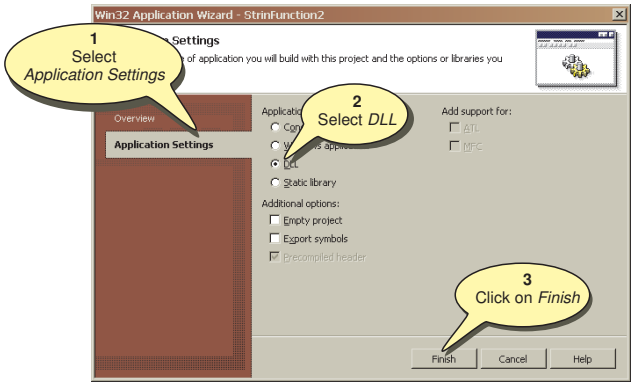
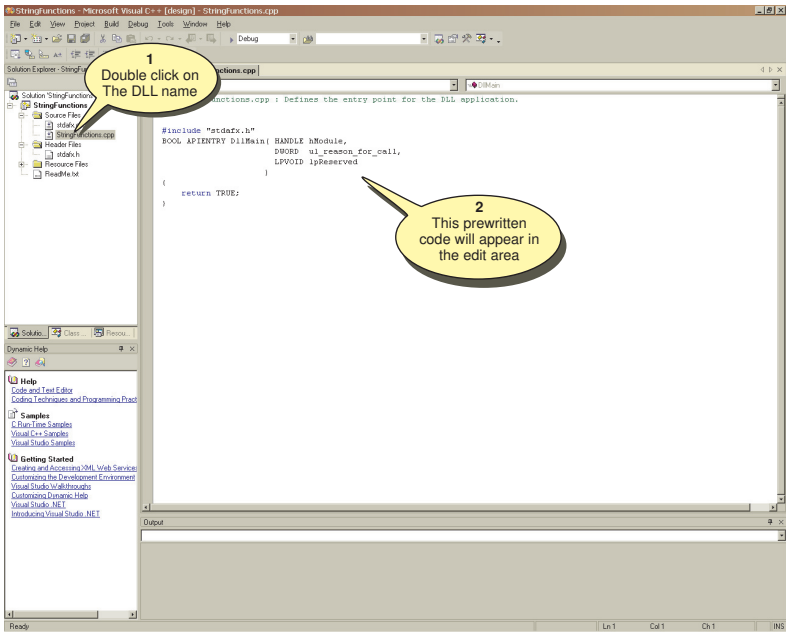


FIG-53.3

The DLL Source File



Adding the Code for New Statements

When the *StringFunctions.cpp* file is opened by double clicking, the editor contains the following lines:

```
// StringFunctions.cpp : Defines the entry point for the DLL
// application.
//
#include "stdafx.h"
BOOL APIENTRY DllMain (    HANDLE hModule,
                          DWORD  ul_reason_for_call,
                          LPVOID lpReserved
                          )
{
    return TRUE;
}
```

Routines which are to be used in other applications need to begin with the rather strange phrase:

```
__declspec(dllexport)
```

but since this is rather awkward to type, we can create a substitute term in a **#define** statement:

```
#define DarkDLL __declspec(dllexport)
```

Now when we write our code, we can use the term *DarkDLL* rather than *__declspec(dllexport)*. Just before compilation begins, the compiler will replace all occurrences of the term *DarkDLL* with the correct term, *__declspec(dllexport)*.

Since we will not be using the additional features of C++, it will be easier if we tell the compiler to treat our code as C rather than C++. To do this we need to add the line

```
extern "C"
```

braces are the symbols
{ }

and then enclose everything we write within a set of braces.

Next we need to write a function for every command we intend to add to DarkBASIC Pro. In this example, we are going to add POS which will return the position of a specified character within a string and OCCURS which will return the number of times a character occurs within a string. Both of these functions were described in Chapter 8.

The code required for the two functions is:

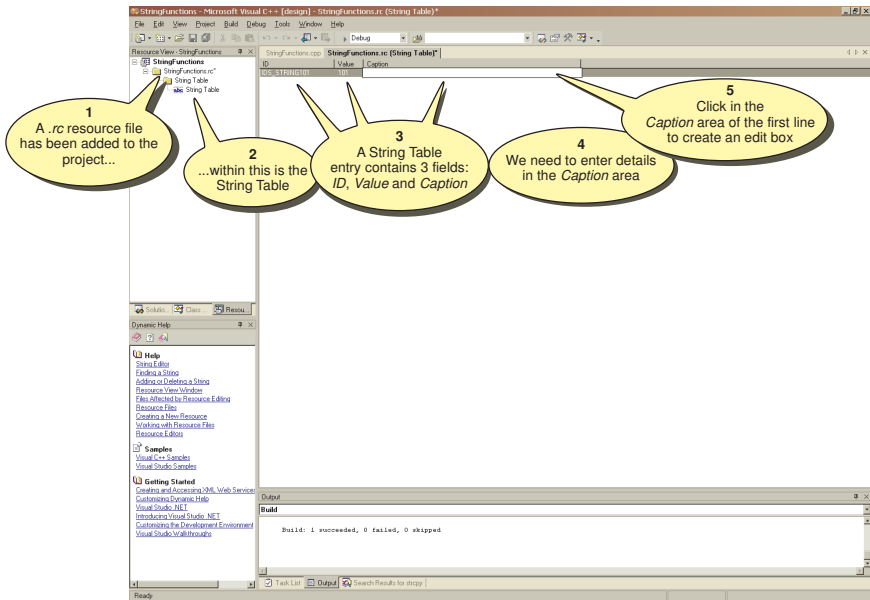
```
DarkDLL int pos(LPSTR s1, LPSTR ch)
{
    int result = -1;
    for(int c = 0; c < strlen(s1); c++)
        if (s1[c] == ch[0])
        {
            result = c+1;
            break;
        }
    return result;
}

DarkDLL int occurs(LPSTR s1, LPSTR ch)
{
    int result = 0;
```


Once *String Table* has been selected, the project files are updated and we need to add an entry to the string table for each of our functions that are included in the DLL (see FIG-53.6).

FIG-53.6

An Entry in the String Table



Constructing the Caption

The caption is made up of several parts with a percent symbol (%) separating each part. These parts are:

The name of the new DarkBASIC Pro statement

This is whatever word or words are to be used to invoke execution of the function. This need not be the same as the function's name. If the function returns a value, the name should end with an opening square bracket ([).

Parameter list

The type of the function's return value and each of its parameters are coded as a letter. The letters used are shown in TABLE-53.1.

TABLE-53.1

Parameter List Codes

Code	Parameter Type
L	integer
F	real (float)
S	string
O	double
R	longint
D	others
0	void

If you haven't enclosed your functions in an **extern "C"** block, C++ creates mangled function names and it's these mangled names that must be used in the string table. You can find out what the mangled name is by searching the DLL file that we will create for the original file name.

The function's name

The name of the function.
In fact, C++ makes up its own name for a function, but by using the **extern "C"** statement we have ensured that the original function name is used. For example, without **extern** the function *occurs* could be called *?occurs@@YAKH*.

The parameters in English

This part lists the parameters of the function in English and can be used to add help for the new instruction in the DarkBASIC Pro environment.

Now we're ready to attempt the caption for our first entry in the string table. Since the first function is called *pos* and we want to use the same term in DarkBASIC Pro, then our statement name will be

```
POS
```

but, since the function returns a value, we need to change this to

```
POS[
```

Next we have a separator, *%*, the return type, and the parameter list. The function returns an integer (L) and takes a string (S) and character (S) as parameters, so the caption now becomes:

```
POS[%LSS
```

Another separator and the function's C++ name give us:

```
POS[%LSS%pos
```

And, finally, a separator and the parameter list in English:

```
POS[%LSS%pos%string,string
```

Activity 53.3

Write down the caption required for the *occurs()* function. Enter the captions for both functions into the string table and close the table.

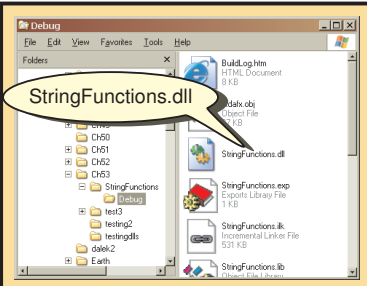
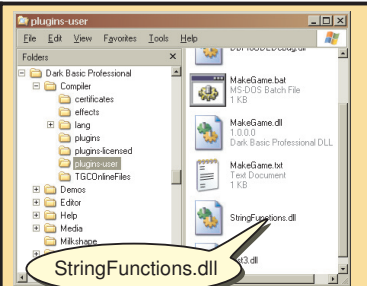
Adding the New Statements to DarkBASIC Pro

With the string table complete, we need to rebuild our code (**Build|BuildSolution**). This creates a file called *StringFunctions.dll* which is placed in a sub-folder named *debug*.

DarkBASIC Pro contains a folder named *Compiler*, and within this is a sub-folder named *plugins-user*. We need to copy *StringFunctions.dll* into this folder. Once this is done, the new statements, POS and OCCURS will be recognised by DarkBASIC Pro (see FIG-53.7).

FIG-53.7

Copying the DLL

		<pre>A\$ = "ABCDEFBBBFRGB" B\$ = "B" PRINT B\$, " appears ", OCCURS(A\$,B\$), " times" PRINT B\$, " first appears at position ", POS(A\$,B\$) WAIT KEY END</pre>
Building the code within C++ will create a file named <i>StringFunctions.dll</i> in the <i>Debug</i> sub-folder.	This file must be copied into <i>Plugins-user</i> in the DarkBASIC Pro's <i>Compiler</i> folder	Now the new statements can be used within DarkBASIC Pro.

Activity 53.4

Carry out the steps explained above and test out the OCCURS and POS statements in a short DarkBASIC Pro program.

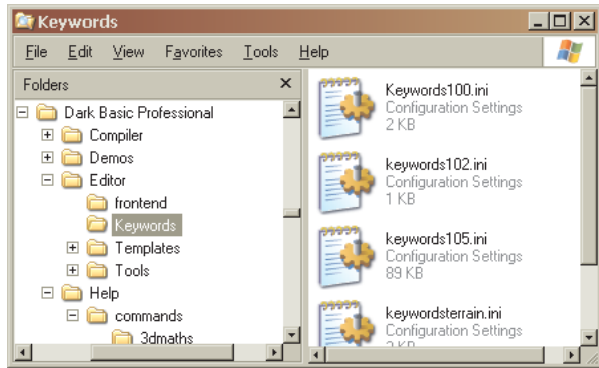
Adding Help

Our new commands aren't fully integrated into DarkBASIC Pro because, even though they function correctly, they are not highlighted in blue and don't bring up a help window when you move the cursor over the statement and press F1. But, with a bit more effort, we can have all that happen too.

The blue highlight is created by having a command listed in one of several special *.ini* files. These files are in DarkBASIC Pro's *Keywords* folder within the *Editor* (see FIG-53.8).

FIG-53.8

The *.ini* Files



Only statements listed in one of these files will appear with a blue highlight in the edit window, so if we want our commands to be highlighted, they must either be added to one of the existing *.ini* files, or placed in a new *.ini* file.

In this case, we'll create a new *.ini* file - that way it will be easier to remove all our changes should that be required.

Activity 53.5

Create a new file in Notepad and add the following lines:

```
OCCURS=commands\extras\OCCURS.htm=(searched,searchedfor)
POS=commands\extras\POS.htm=(searched,searchedfor)
```

Save the file as *myfunctions.ini*.

Load up the DarkBASIC Pro program you created in Activity 53.4. Are the terms OCCURS and POS now highlighted in blue?

Although we've created the highlighting, we still need to create a help page for each statement. In fact, the *.ini* file actually lists details of the help file name and where it is to be found. For example, in the line

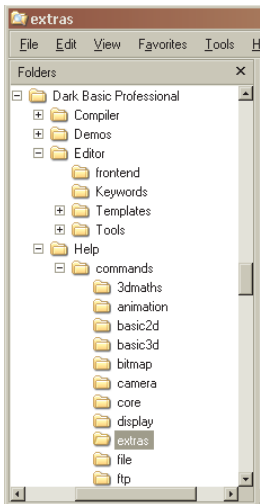
```
OCCURS=commands\extras\OCCURS.htm=(searched,searchedfor)
```

the term *commands\extras\OCCURS.htm* tells DarkBASIC Pro that the help page for the OCCURS statement is in a file named *OCCURS.htm* which is to be found

in the *extras* sub-folder within the *commands* folder which is itself within DarkBASIC Pro's *Help* folder (see FIG-53.9)

FIG-53.9

The *extras* Folder



Activity 53.6

The *extras* folder doesn't yet exist, so it needs to be created.

Create an *extras* sub-folder within Help's *commands* folder as shown above.

Now we need to create each of the help files. These are just HTML files giving details of a command. The code for *OCCURS.htm* is shown below:

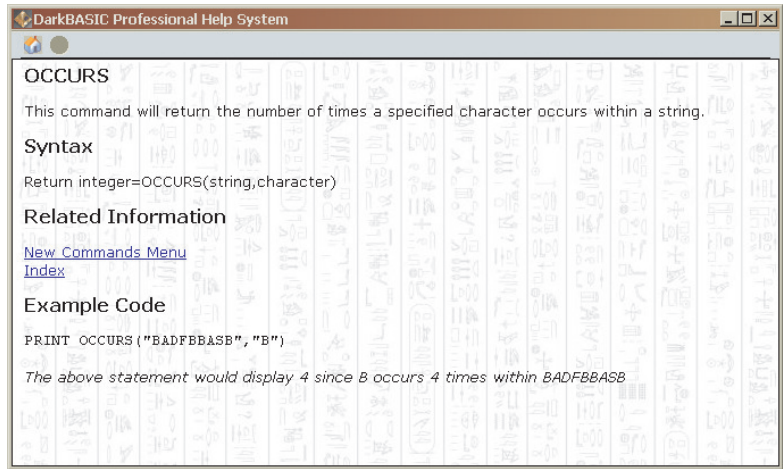
```
<BODY COLOR=black BACKGROUND="..\gfx/backdrop.jpg"
BGPROPERTIES=FIXED><FONT FACE=Verdana SIZE=2>
<font face=Verdana size=2><font size=3><b>
OCCURS
</b></font>
<BR>
<p>
This command will return the number of times a specified character occurs within a
string.
</p>
<font size=3><b>
Syntax
</b></font>
<p>Return integer=OCCURS(string,character)<BR>
</p><ul>
</ul>
<font size=3><b>
Related Information
</b></font>
<p>
<a href="..\extras.htm">New Commands Menu<BR>
</a>
<a href="..\index.htm">Index</a><BR>
</p>
<font size=3><b>
Example Code
</b></font>
<p>
<code>
PRINT OCCURS("BADFBBASB","B")
</code>
<br/><br/>
<i>The above statement would display 4 since B occurs 4 times within BADFBBASB
</i>
```

```
</p>
</FONT></BODY>
```

This will create the page shown in FIG-53.10.

FIG-53.10

The Help Page for
OCCURS



The code for the second file, *POS.htm*, is:

```
<BODY COLOR=black BACKGROUND="..\gfx/backdrop.jpg"
BGPROPERTIES=FIXED><FONT FACE=Verdana SIZE=2>
<font face=Verdana size=2><font size=3><b>
POS
</b></font>
<BR>
<p>
This command will return the first position at which a specified character occurs within a
string.
If the character does not appear within the string, the value -1 is returned.
</p>
<font size=3><b>
Syntax
</b></font>
<p>Return integer=POS(string,character)<BR>
</p><ul>
</ul>
<font size=3><b>
Related Information
</b></font>
<p>
<a href="..\extras.htm">New Commands Menu<BR>
</a>
<a href="..\index.htm">Index</a><BR>
</p>
<font size=3><b>
Example Code
</b></font>
<p>
<code>
PRINT POS("BADFBBASB","B")
</code>
<br/><br/>
<i> The above statement would display 1 since B first occurs at position 1 within
BADFBBASB </i>
<br/><br/><br/>
<code>
<pre>
A$ = "ABCDEF"
B$ = "X"
result = POS(A$,B$)
</pre>
</code>
```



```

result <i> would be set to -1 since X does not appear within ABCDEF. </i>
</p>
</FONT></BODY>

```

Activity 53.7

Create both *OCCURS.htm* and *POS.htm*, containing the code given and save them in the *extras* folder.

It is usual for all the instructions that belong to a specific group (for example, camera statements, lighting statements, etc.) to have an HTML file listing each of the commands in that group with links to their individual help pages. In fact, our two new pages already have links to the *extras* command list page - even though we haven't created it yet!

This last help-related file, *extras.htm*, has the following code:

```

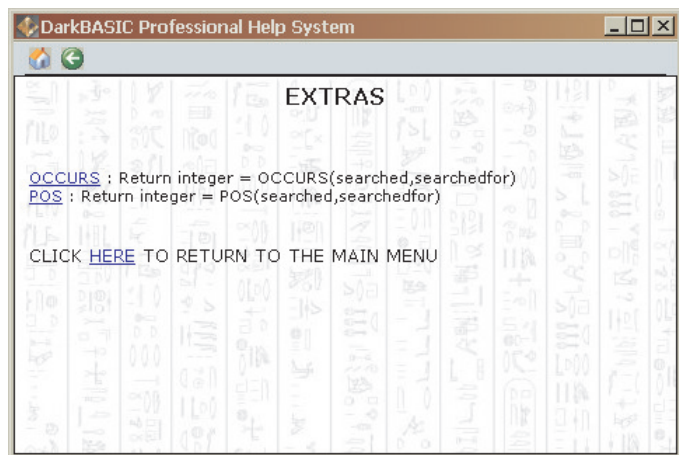
<BODY COLOR=black BACKGROUND="gfx/backdrop.jpg"
BGPROPERTIES=FIXED><FONT FACE=Verdana SIZE=2><CENTER>
<FONT SIZE=3><B>EXTRAS</B></FONT></CENTER><BR>
<BR>
<PRE><FONT FACE=Verdana SIZE=2>
<a href="extras\OCCURS.htm">OCCURS</a> : Return integer =
OCCURS(searched,searchedfor)
<a href="extras\POS.htm">POS</a> : Return integer = POS(searched,searchedfor)
</FONT>
</PRE><BR/>
CLICK <A HREF="..\main.htm">HERE</A> TO RETURN TO THE MAIN MENU<BR>
<BR/>
</FONT></BODY>

```

and creates the page shown in FIG-53.11.

FIG-53.11

A Group Help File



Activity 53.8

Create *extras.htm* and save it in the DarkBASIC Pro's *commands* folder.

At last we have a completely integrated set of new commands.

Adding More New Commands

Some new commands are going to prove to be slightly more difficult to create than OCCURS and POS.

Functions that Return Real Values

Functions which return real values should in theory be written something like:

```
DarkDLL float returnfloat()
{
    float result = 12.3F;
    return result;
}
```

But although the code will compile, the value returned bears no resemblance to 12.3. To make the whole thing work as expected, we need to start by changing the return type to DWORD

```
DarkDLL DWORD returnfloat()
```

and then the returned value must be modified to

```
return *(DWORD*)&result;
```

which means the whole routine is actually coded as:

```
DarkDLL DWORD returnfloat()
{
    float result = 12.3F;
    return *(DWORD*)&result;
}
```

The string table caption for this routine could be:

```
RETURNREAL[%F%returnfloat
```

Notice on this occasion the term used in DarkBASIC Pro is RETURNREAL, while the function is actually called *returnfloat*. Also, since there are no parameters, the final part of the entry is missing.

Functions that Return Strings

When we need to return a string from a function, things get seriously weird! To start with we need to include a file named *globstruct.h* using the line:

```
#include "globstruct.h"
```

You'll find this file in some examples embedded in DarkBASIC Pro in the folder

```
program files\dark basic professional\help\documents\files\Third Party Commands
↳ |TESTCOMMANDS2
```

The file must be copied into the folder containing your DLL source code.

Unfortunately, *globstruct.h* makes references to other header files from the DirectX development kit. If you don't have this, you won't be able to compile your program. However, assuming only simple functions are required in your DLL we can doctor

Activity 53.9

Find *globstruct.h* and copy it into the DLL source code folder.

Make the following changes to the file's code:

Comment out the line `#include "D3dx9tex.h"`

Change the lines

```
// Bitmap and Surface Data (for drawing offscreen)
int                                iCurrentBitmapNumber;
LPDIRECT3DTEXTURE9                pCurrentBitmapTexture;
LPDIRECT3DSURFACE9                pCurrentBitmapSurface;
LPDIRECT3DSURFACE9                pHoldBackBufferPtr;
LPDIRECT3DSURFACE9                pHoldDepthBufferPtr ;
to
// Bitmap and Surface Data (for drawing offscreen)
int                                iCurrentBitmapNumber;
void*                              pCurrentBitmapTexture;
void*                              pCurrentBitmapSurface;
void*                              pHoldBackBufferPtr;
void*                              pHoldDepthBufferPtr;
```

Also change the line

```
char                                pEXEUnpackDirectory[_MAX_PATH];
to
char                                pEXEUnpackDirectory[260];
```

Back in the code for the DLL, we need to create a global variable immediately after the new `#include` statement:

```
GlobStruct* g_pGlob = NULL;
```

Activity 53.10

In your DLL source file, add the lines

```
#include "globstruct.h"
GlobStruct* g_pGlob = NULL;
```

immediately after

```
#include "stdafx.h"
```

A function that returns a string has to be written as returning a `DWORD` type. But it also needs to use a phantom parameter of the same type:

```
DarkDLL DWORD returnstring(DWORD olds)
```

Within the routine we need to treat the parameter as a pointer and delete any space it is referencing with the lines:

```

if (olds)
    g_pGlob->CreateDeleteString((DWORD*) &olds, 0);

```

The string which is going to contain the result needs to be set up with lines such as:

```

LPSTR strresult=NULL;
g_pGlob->CreateDeleteString((DWORD*) &strresult, 10);

```

The value at the end of the second line is the size of the string, taking into account that strings in C++ must be assigned an extra character position for the terminating null character.

Now, as long as we cast *strresult* to LPSTR we can treat it as a normal C++ string. For example, we could store the character sequence ABCD in the string using the line:

```

strcpy((LPSTR) strresult, "ABCD");

```

To return the string from the function we need to end with a line such as

```

return (DWORD) strresult;

```

casting the result to a DWORD type.

So, the code for the complete function would be:

```

DarkDLL DWORD returnstring(DWORD olds)
{
    if (olds)
        g_pGlob->CreateDeleteString((DWORD*) &olds, 0);
    LPSTR strresult=NULL;
    g_pGlob->CreateDeleteString((DWORD*) &strresult, 5);
    strcpy((LPSTR) strresult, "ABCD");
    return (DWORD) strresult;
}

```

The *Caption* in the string table holds another surprise. We'll start by using a DarkBASIC Pro type name for the new instruction

```

ABCD$[

```

and since it is designed to return a string, this expands to

```

ABCD$[%S

```

but the phantom parameter is ignored, so we continue directly to the function name:

```

ABCD$[%S%returnstring

```

Because there are no parameters, the final comment section of the string can be omitted or the term *void* used.

Activity 53.11

Add *returnstring()* to your DLL file along with an entry in the string table.

Try out the new statement in a DarkBASIC Pro program.

After testing, remove the *returnstring()* function and its string table entry from your C++ project.

More String Handling Functions

More practical string-handling functions are given in LISTING-53.1. Again these are based on the string functions described in Chapter 8. The functions are:

INSERT\$(A\$,B\$,p)	which returns a string created by embedding string <i>B\$</i> within <i>A\$</i> starting at position <i>p</i> .
DELETE\$(A\$,p,n)	which returns a string formed by deleting <i>n</i> characters from <i>A\$</i> starting with the character at position <i>p</i> .
REPLACE\$(A\$,B\$,p)	which returns a string formed by replacing the <i>p</i> th character in <i>A\$</i> with the character <i>B\$</i> .

LISTING-53.1

C++ Code for the New Statements

```
DarkDLL DWORD insertstr(DWORD olds, DWORD s1, DWORD s2, int post)
{
    if(olds)
        g_pGlob->CreateDeleteString ( (DWORD*)&olds, 0 );
    if (post < 1)
        post = 1;
    if (post > strlen((LPSTR)s1)+1)
        post = strlen((LPSTR)s1)+1;
    LPSTR strresult=NULL;
    g_pGlob->CreateDeleteString((DWORD*)&strresult,
        strlen((LPSTR)s1)+strlen((LPSTR)s2)+1 );
    strncpy(strresult, (LPSTR)s1, post-1);
    strresult[post-1]='\0';
    strcat(strresult, (LPSTR)s2);
    strcat(strresult, &((LPSTR)s1)[post-1]);
    return (DWORD)strresult;
}

DarkDLL DWORD deletestr(DWORD olds, DWORD s, int start, int num)
{
    if(olds)
        g_pGlob->CreateDeleteString ( (DWORD*)&olds, 0 );
    LPSTR strresult=NULL;
    if(start < 1 || start > strlen((LPSTR)s) || num<1)
    {
        g_pGlob->CreateDeleteString((DWORD*)&strresult,
            strlen((LPSTR)s)+1);
        strcpy(strresult, (LPSTR)s);
    }
    else
    {
        g_pGlob->CreateDeleteString((DWORD*)&strresult, num+1);
        strncpy(strresult, (LPSTR)s, start-1);
        strresult[start-1] = '\0';
        if(start+num > strlen((LPSTR)s))
            num = strlen((LPSTR)s)-start;
        strcat(strresult, &((LPSTR)s)[start+num-1]);
    }
    return (DWORD)strresult;
}
```

continued on next page

LISTING-53.1

(continued)

C++ Code for the New
Statements

```
DarkDLL DWORD replacechr(DWORD olds, DWORD s, DWORD ch, int post)
{
    if(olds)
        g_pGlob->CreateDeleteString ( (DWORD*)&olds, 0 );
    LPSTR strresult=NULL;
    g_pGlob->CreateDeleteString((DWORD*)&strresult,
        strlen((LPSTR)s)+1);
    if(post < 1 || post > strlen((LPSTR)s) || strlen((LPSTR)ch)>1)
        strcpy(strresult, (LPSTR)s);
    else
    {
        strncpy(strresult, (LPSTR)s, post-1);
        strresult[post-1] = ((LPSTR)ch)[0];
        strresult[post]='\0';
        strcat(strresult, &((LPSTR)s)[post]);
    }
    return (DWORD)strresult;
}
```

Activity 53.12

Add the three functions given in LISTING-53.1 to your DLL source code.

Add appropriate entries in the string table.

Test the new statements in a DarkBASIC Pro program.

Summary

- A Dynamic Link Library (DLL) is a file containing a collection of functions.
- DLLs can be created using a variety of languages.
- In Visual Studio's C++, the core of a DLL can be created automatically.
- Functions within a DLL that are designed to be used elsewhere should be preceded by the term `__declspec(dllexport)`.
- By including all functions within an `extern "C"` block, function names used by the operating system match those given in the source code.
- Functions not included in an `extern` block have new names allocated by the C++ compiler.
- C++ generated names are known as mangled or decorated function names.
- The mangled name of a function can be found within the DLL file generated by the C++ compiler.
- Return values and parameters used when coding DLL functions must match those required by DarkBASIC Pro.
- Real parameters should be declared as type `DWORD`.
- String parameters should be `LPSTR` or `DWORD`.
- A string table must be added to the DLL project.

- Each entry in a string table gives the name to be used when the function is called from DarkBASIC Pro, the function's return type and parameters, the actual function name, and a comment listing the parameter types.
- New commands can be highlighted in blue in the DarkBASIC Pro editor by adding a *.ini* file to the editor's Keywords folder.
- Entries in a *.ini* file give the name of the command, the path to the corresponding HTML help file and a parameter list.
- An HTML file should be added to the appropriate folder for each new command.
- If a function returns a string the DLL should include *globstruct.h*.
- Elements of *globstruct.h* must be edited or other *.h* files from DirectX SDK must be made available.
- A function that returns a string must include a phantom DWORD parameter.
- The phantom parameter is omitted from the string table and ignored when calling the function.

Using Standard DLLs

Introduction

DLLs don't have to be used just to create new statements for DarkBASIC Pro. It is also possible to make use of other DLLs simply because they contain functions which would be difficult or impossible to implement in DarkBASIC Pro.

However, any functions within a DLL using real or string values must still be written using the techniques explained in the previous section of this chapter.

LISTING-53.2 shows the contents of a DLL file named *functions.dll*.

LISTING-53.2

Standard DLL Functions

```
// Functions.cpp : Defines the entry point for the DLL
// application.
//

#include "stdafx.h"
BOOL APIENTRY DllMain( HANDLE hModule,
                      DWORD  ul_reason_for_call,
                      LPVOID lpReserved
                      )
{
    return TRUE;
}

#define DarkDLL _declspec(dllexport)
extern "C"
{
    DarkDLL LONGLONG factorial(int v)
    {
        LONGLONG result = 1;
        for(int c = 2; c <= v; c++)
            result *=c;
        return result;
    }

    DarkDLL DWORD reciprocal(int v)
    {
        float result = 1/v;
        return *(DWORD*)&result;
    }
}
```

The file contains two functions.

The first, *factorial()*, returns the factorial of a specified value. A factorial of a value is the result obtained by multiplying the value by every integer from 2 up to itself. For example, factorial 5 (written in mathematics as 5!) is 2*3*4*5, or, 120.

The second function, *reciprocal()*, returns the value of 1 divided by the number given. For example, the reciprocal of 5 is 1/5, or 0.2.

Activity 53.13

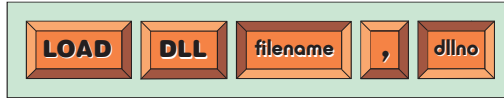
Create a new C++ Win32 DLL project and type in the code given in LISTING-53.2. Check that the code compiles correctly.

The LOAD DLL Statement

To use the functions in a DLL which have not been added as new statements in DarkBASIC Pro requires the DLL to first be loaded into our program. This is done using the LOAD DLL statement which has the format shown in FIG-53.12.

FIG-53.12

The LOAD DLL Statement



In the diagram:

filename

is a string giving the name of the file containing the DLL. This file should be in the current DarkBASIC Pro project's folder.

dllno

is an integer value giving the ID to be assigned to the DLL.

For example, we could load *functions.dll* into our current project using the line:

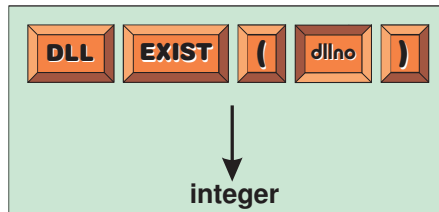
```
LOAD DLL "functions.dll",1
```

The DLL EXIST Statement

It is possible that a DLL will not be successfully loaded - normally because we've forgotten to copy the DLL file into our current project's folder. To check that it has been loaded successfully, we need to use the DLL EXIST statement which returns 1 if the DLL specified has been loaded, and zero, if there's been a problem. This statement has the format shown in FIG-53.13.

FIG-53.13

The DLL EXIST Statement



In the diagram:

dllno

is an integer value specifying the ID of the DLL to be checked.

Our program should check for the DLL being loaded immediately after the LOAD DLL statement:

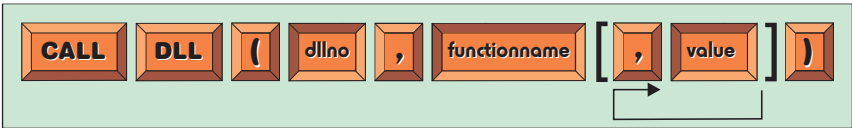
```
LOAD DLL "functions.dll",1
IF DLL EXIST(1)=0
    PRINT "DLL not found"
ENDIF
```

The CALL DLL Statement

With a DLL successfully loaded, we are free to execute any of the functions within the DLL using the CALL DLL statement which has the format shown in FIG-53.14.

FIG-53.14

The CALL DLL
Statement



In the diagram:

- dllno* is an integer value giving the ID of the DLL containing the function to be executed.
- functionname* is a string giving the name of the function to be executed.
- value* represents any parameter that is to be passed to the function. There can be as many comma-separated values as required by the function.

For example, we could execute the *factorial()* function in our DLL using the line

```
CALL DLL(1,"factorial",5)
```

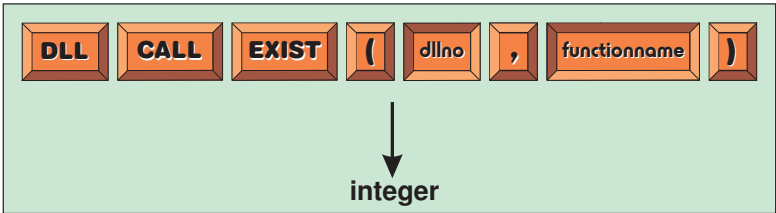
which would return the value of 5!

The DLL CALL EXIST Statement

If we're going to play extra safe - as we should - we should check that a function of the name specified actually exists within the DLL before we attempt to execute that function. This can be done using the DLL CALL EXIST statement whose format is given in FIG-53.15.

FIG-53.15

The DLL CALL EXIST
Statement



In the diagram:

- dllno* is an integer giving the ID of the DLL containing the function being checked.
- functionname* is a string giving the name of the function to be checked.

The statement returns 1 if the specified function exists within the DLL, otherwise zero is returned.

The DELETE DLL Statement

When we've finished with a DLL, we can ask the operating system to remove the DDL from memory. This is done using the DELETE DLL statement whose format is shown in FIG-53.16.

FIG-53.16

The DELETE DLL
Statement



In the diagram:

dllno is an integer value giving the ID of the DLL to be removed from memory.

In fact, the DLL is only removed if no other program is using it.

The program in LISTING-53.3 demonstrates the use of a standard DLL by executing the *factorial()* and *reciprocal()* routines in the *functions.dll*.

LISTING-53.3

Using a Standard DLL

```
REM *** Load the DLL ***
LOAD DLL "Functions.dll",1
REM *** IF it loaded THEN ***
IF DLL EXIST(1)
  REM *** IF factorial exists, execute it ***
  IF DLL CALL EXIST(1,"factorial")
    result = CALL DLL(1,"factorial",5)
    PRINT "5! = ",result
  ENDIF
  REM *** IF reciprocal exists, execute it ***
  IF DLL CALL EXIST (1,"reciprocal")
    result# = CALL DLL(1,"reciprocal",10)
    PRINT " 1/10 = ",result#
  ENDIF
  REM *** Delete the DLL ***
  DELETE DLL 1
ELSE
  REM *** Display message if DLL not found ***
  PRINT "DLL not found"
ENDIF
REM *** End program ***
WAIT KEY
END
```

Activity 53.14

Type in the program in LISTING-53.3 (*dll03.dbpro*).

Copy the *functions.dll* file into the *dll03* folder.

Run the program and check that the two functions are executed correctly.

Summary

- DLLs not designed to create new DarkBASIC Pro instructions can also be called from a DarkBASIC Pro program.
- The functions in such a DLL must conform to DarkBASIC Pro's method of dealing with real and string values.
- No string table is required for these DLLs.
- Use LOAD DLL to load a required DLL into a program.
- Use DLL EXIST to check that a DLL has been loaded.

- Use CALL DLL to call a function within a DLL.
- Use DLL CALL EXIST to check that a function of a specified name exists within a DLL.
- Use DELETE DLL to delete a DLL from memory.

At last, we've reached the end of two massive volumes covering most of DarkBASIC Pro's statements. I hope you have found the effort worthwhile and are inspired to start work on a game of your own.

But remember, there's a lot more you can learn from keeping in touch on The Game Creators forums where you will find people with an amazing amount of experience who are more than willing to help with any problems you have.

If you find any mistakes in this text or have any ideas about extra topics that might be covered, I would genuinely like to hear from you. You can contact me at alistair@digital-skills.co.uk. You'll also find supplementary material on the DarkBASIC Pro section of our website at www.digital-skills.co.uk.

Solutions

Activity 53.1

No solution required.

Activity 53.2

The code in *StringFunctions.cpp* should be:

```
// StringFunctions.cpp : Defines the entry
// point for the DLL application.
//

#include "stdafx.h"

BOOL APIENTRY DllMain
(
    HANDLE hModule,
    DWORD ul_reason_for_call,
    LPVOID lpReserved
)
{
    return TRUE;
}

#define DarkDLL __declspec(dllexport)

extern "C"
{
    DarkDLL int pos(LPSTR s1, LPSTR ch)
    {
        int result = -1;
        for(int c = 0; c < strlen(s1); c++)
            if (s1[c] == ch[0])
            {
                result = c+1;
                break;
            }
        return result;
    }

    DarkDLL int occurs(LPSTR s1, LPSTR ch)
    {
        int result = 0;
        for(int c = 0; c < strlen(s1); c++)
            if (s1[c] == ch[0])
                result++;
        return result;
    }
}
```

Activity 53.3

The string table caption for *occurs()* should be:

```
OCCURS[%LSS%occurs%string, string
```

Activity 53.4

A very simple DarkBASIC Pro program such as

```
PRINT POS("ABCD", "C")
PRINT OCCURS("ABBBCCDBBDB", "B")
WAIT KEY
END
```

should act as a basic test of the new statements.

Activity 53.5

The new statements, POS and OCCURS, should now be highlighted in blue.

Activity 53.6

No solution required.

Activity 53.7

No solution required.

Activity 53.8

No solution required.

Activity 53.9

No solution required.

Activity 53.10

No solution required.

Activity 53.11

The code in *StringFunctions.cpp* should now be:

```
// StringFunctions.cpp : Defines the entry
// point for the DLL application.
//

#include "stdafx.h"
#include "globstruct.h"
GlobStruct* g_pGlob = NULL;

BOOL APIENTRY DllMain
(
    HANDLE hModule,
    DWORD ul_reason_for_call,
    LPVOID lpReserved
)
{
    return TRUE;
}

#define DarkDLL __declspec(dllexport)

extern "C"
{
    DarkDLL int pos(LPSTR s1, LPSTR ch)
    {
        int result = -1;
        for(int c = 0; c < strlen(s1); c++)
            if (s1[c] == ch[0])
            {
                result = c+1;
                break;
            }
        return result;
    }

    DarkDLL int occurs(LPSTR s1, LPSTR ch)
    {
        int result = 0;
        for(int c = 0; c < strlen(s1); c++)
```

```

        if (s1[c] == ch[0])
            result++;
        return result;
    }

DarkDLL DWORD returnstring(DWORD olds)
{
    if (olds)
        g_pGlob->CreateDeleteString
            ↳ ((DWORD*)&olds, 0);
    LPSTR strresult=NULL;
    g_pGlob->CreateDeleteString
        ↳ ((DWORD*)&strresult, 5);
    strcpy((LPSTR)strresult, "ABCD");
    return (DWORD)strresult;
}
}

```

A DarkBASIC Pro program such as

```

PRINT ABCD$()
WAIT KEY
END

```

will call the new routine and display ABCD.

Activity 53.12

String table entries for the new functions should be:

```

INSERT$[%SSSL%insert%string,string,int
DELETE$[%SSLL%deletestr%string,int,int
REPLACE$[%SSSL%replacechr%string,character,int

```

A DarkBASIC Pro program such as

```

PRINT INSERT$ ("ABCD", "XYZ", 2)
PRINT DELETE$ ("ABCD", 2, 2)
PRINT REPLACE$ ("ABCD", "X", 3)
WAIT KEY
END

```

should give an indication that the new statements are working.

Activity 53.13

No solution required.

Activity 53.14

No solution required.

INDEX

3D primitives	749	Constructive Solid Geometry (CSG)	788
3D Vectors	1350	Contact joint	1340
column vectors	1350	CONVERT OBJECT FVF	1386
direction	1350	CONTROL CAMERA USING ARROWKEYS	856
elements	1350	Coordinates system	744
magnitude	1350	COPY MATRIX4	1367
row vectors	1350	COPY MEMBLOCK	1287
4 by 4 Matrices	1365	COPY VECTOR3	1353
4D Vectors	1363	CREATE NET GAME	1394
homogeneous coordinates	1363	CREATE NET PLAYER	1412
w	1363	CREATE VERTEX SHADER FROM FILE	1384
__declspec(dllexport)	1448	Creating New DBPro Statements	1446
		CROSS PRODUCT VECTOR3	1360
		Culling	784

A

Accessing Memory	1280
ADD LIMB	917
ADD MATRIX	1368
ADD MESH TO VERTEXDATA	1265
ADD VECTOR3	1356
Advanced Camera Techniques	871
Advanced Terrain Statements	1186
Alpha channel	750, 820
Ambient Lighting	886
APPEND OBJECT	970
AUTOCAM	851
AUTOMATIC CAMERA COLLISION	858
AUTOMATIC OBJECT COLLISION	1082
Axes	744

B

BACKDROP	864
Binary Space Partitioning	1164
Bit	1280
Bounding box	1071
BSP COLLISION	1172
BSP COLLISION HIT	1172
BUILD TERRAIN	1188
Bump map	1038
Bytes	1280

C

CALL DLL	1463
Cameras	751, 836
CAMERA ANGLE	844
CAMERA POSITION	844
Castle model	822
CHANGE MESH	931
CHANGE MESH FROM MEMBLOCK	1310
CHECK LIMB LINK	947
CLEAR CAMERA VIEW	869
Client/server	1390
CLONE OBJECT	778
Collision box	1071
Collision sphere	1071
COLOR AMBIENT LIGHT	889
COLOR BACKDROP	864
COLOR LIGHT	890
COLOR LIMB	922
COLOR OBJECT	815
COLOR PARTICLES	1107
Column vectors	1350

D

DBO file format	948
DELETE BSP	1173
DELETE CAMERA	866
DELETE DLL	1464
DELETE LIGHT	890
DELETE MATRIX4	1372
DELETE MEMBLOCK	1286
DELETE MESH	915
DELETE MESH FROM VERTEXDATA	1271
DELETE OBJECT	777
DELETE OBJECTS	777
DELETE OBJECT COLLISION BOX	1082
DELETE PARTICLES	1104
DELETE PIXEL SHADER	1386
DELETE TERRAIN	1181
DELETE VECTOR3	1353
Dereference	1282
Directional lighting	886
DIVIDE MATRIX4	1368
DIVIDE VECTOR3	1355
DLL	1446
DLL CALL EXIST	1464
DLL EXIST	1463
DOT PRODUCT VECTOR3	1357
Doors	935
Dungeon Torch	1122

E

Edge	749
EFFECT EXIST	1378
Elevator model	948
emitter	1102
extern "C"	1448

F

FADE OBJECT	817
File formats	963
FILL MATRIX	1222
FIRE PARTICLES	1119
FIX OBJECT PIVOT	773
FOG	901
FOG COLOR	902
FOG DISTANCE	902
Frame rate	750
FREE NET GAME	1411

MAKE MESH FROM OBJECT	912	OBJECT COLLISION CENTER	1075
MAKE OBJECT	914	OBJECT COLLISION RADIUS	1075
MAKE OBJECT BOX	757	OBJECT EXIST	779
MAKE OBJECT COLLISION BOX	1077	OBJECT FRAME	972
MAKE OBJECT CONE	760	OBJECT HIT	1069
MAKE OBJECT CYLINDER	759	OBJECT IN SCREEN	987
MAKE OBJECT CUBE	756	OBJECT INTERPOLATION	972
MAKE OBJECT FROM LIMB	919	OBJECT LOOPING	971
MAKE OBJECT PLAIN	760	OBJECT PLAYING	971
MAKE OBJECT SPHERE	758	OBJECT POSITION	780
MAKE OBJECT TERRAIN	1186	OBJECT SCREEN	982
MAKE OBJECT TRIANGLE	761	OBJECT SIZE	781, 973
MAKE PARTICLES	1102	OBJECT SPEED	972
MAKE SNOW PARTICLES	1118	OBJECT VISIBLE	780
MAKE SOUND FROM MEMBLOCK	1305	ODE	1318
MAKE STATIC COLLISION BOX	1087	ODE ADD FORCE	1338
MAKE TERRAIN	1180	ODE COLLISION GET MESSAGE	1336
MAKE VECTOR3	1351	ODE COLLISION MESSAGE EXISTS	1336
Manipulating Veticies	1246	ODE CREATE DYNAMIC BOX	1318
Map	1164	ODE CREATE DYNAMIC CYLINDER	1322
Matrices	1212	ODE CREATE DYNAMIC TRIANGLE MESH	1324
MATRIX EXIST	1237	ODE CREATE STATIC BOX	1321
MATRIX POSITION	1230	ODE CREATE STATIC TRIANGLE MESH	1325
MATRIX TILE COUNT	1228	ODE DESTROY OBJECT	1334
MATRIX TILES EXIST	1228	ODE END	1319
MATRIX WIREFRAME STATE	1220	ODE GET BODY HEIGHT	1335
MAXIMIZE VECTOR3	1359	ODE GET BODY LINEAR VELOCITY	1335
MEMBLOCK	1285	ODE GET OBJECT	1336
MEMBLOCK EXIST	1286	ODE GET OBJECT ANGULAR VELOCITY	1338
Memory Blocks		ODE GET OBJECT VELOCITY	1337
Media	1297	ODE MAKE DYNAMIC SPHERE	1322
Strings	1288	ODE SET ANGULAR VELOCITY	1331
Merging 3D primitives	788	ODE SET BODY MASS	1332
MESH EXIST	915	ODE SET BODY ROTATION	1332
Meshes	912	ODE SET CONTACT FDIR1	1328
MINIMIZE VECTOR3	1360	ODE SET LINEAR VELOCITY	1328
Mipmaps	799	ODE SET WORLD CFM	1327
MOVE CAMERA	837	ODE SET WORLD ERP	1326
MOVE OBJECT	764	ODE SET WORLD GRAVITY	1320
MOVE OBJECT distance	772	ODE SET WORLD STEP	1325
Multiple cameras	863	ODE START	1319
MULTIPLY MATRIX4	1369	ODE UPDATE	1319
MULTIPLY VECTOR3	1354	OFFSET LIMB	920

N

NET BUFFER SIZE	1408
NET GAME EXISTS	1413
NET GAME LOST	1413
NET GAME NOW HOSTING	1411
NET MESSAGE	1402
NET MESSAGE EXISTS	1402
NET MESSAGE PLAYER FROM	1403
NET MESSAGE PLAYER TO	1403
NET MESSAGE TYPE	1406
NET PLAYER CREATED	1409
NET PLAYER DESTROYED	1409
Network	1390
Networked games	1390
NORMALIZE VECTOR3	1358
Normals	752
surface	752
vertex	753

OBJECT ANGLE	781
OBJECT COLLISION	1070

P

Packets	1392
Particles	1102
PARTICLES EXIST	1114
PARTICLES POSITION	1115
Peer-to-peer	1390
PERFORM CHECKLIST FOR EFFECT	1379
PERFORM CSG DIFFERENCE	790
PERFORM CSG INTERSECTION	790
PERFORM CSG UNION	788
PERFORM CHECKLIST FOR NET CONNECTIONS	1391
PERFORM CHECKLIST FOR NET PLAYERS	1397
PERFORM CHECKLIST FOR NET SESSIONS	1395
PERFORM CHECKLIST FOR OBJECT LIMBS	944
PICK SCREEN	986
PICK OBJECT	983
PICK VECTOR	985
PITCH CAMERA	842
PITCH OBJECT	770
Pixel shader	1376
Planes	745
PLAY ANIMATION TO IMAGE	805
PLAY OBJECT	965
POINT CAMERA	838
POINT LIGHT	893

O

POINT OBJECT	771	SET CARTOON SHADING ON	1054
Points	746	SET CUBE MAPPING ON	1042
Pointers	1280	SET CURRENT CAMERA	865
Polygon	749	SET DETAIL MAPPING ON	811
POSITION CAMERA	836	SET DIRECTIONAL LIGHT	892
POSITION LIGHT	891	SET EFFECT CONSTANT	1383
POSITION MATRIX	1229	SET EFFECT ON	1380
POSITION OBJECT	762	SET EFFECT TECHNIQUE	1383
POSITION PARTICLE EMISSIONS	1105	SET EFFECT TRANSPOSE	1384
POSITION PARTICLES	1104	SET GLOBAL COLLISION	1071
POSITION TERRAIN	1182	SET GLOBAL OBJECT CREATION	785
PREPARE MATRIX TEXTURE	1220	SET GLOBAL SHADOW COLOR	1050
PROCESS BSP COLLISION	1171	SET GLOBAL SHADOW SHADES	1050
		SET GLOBAL SHADOWS	1048
		SET IDENTITY MATRIX4	1366
		SET INDEXDATA	1270
		SET LIGHT MAPPING ON	1035
		SET LIGHT RANGE	891
		SET LIGHT TO OBJECT ORIENTATION	897
		SET LIGHT TO OBJECT POSITION	895
		SET LIMB EFFECT	1381
		SET LIMB SMOOTHING	934
		SET MATRIX	1235
		SET MATRIX HEIGHT	1215
		SET MATRIX NORMAL	1234
		SET MATRIX PRIORITY	1232
		SET MATRIX TILE	1223
		SET MATRIX WIREFRAME	1219
		SET NET CONNECTION	1392
		SET OBJECT AMBIENCE	1029
		SET OBJECT COLLISION	1070
		SET OBJECT COLLISION TO BOXES	1076
		SET OBJECT COLLISION TO POLYGON	1076
		SET OBJECT COLLISION TO SPHERE	1074
		SET OBJECT CULL	784
		SET OBJECT DIFFUSE	1030
		SET OBJECT EMISSIVE	1031
		SET OBJECT EFFECT	1379
		SET OBJECT FILTER	813
		SET OBJECT FOG	903
		SET OBJECT FRAME	968
		SET OBJECT INTERPOLATION	969
		SET OBJECT LIGHT	1034
		SET OBJECT ROTATION	769
		SET OBJECT RADIUS	1074
		SET OBJECT SPECULAR	1030
		SET OBJECT SPECULAR POWER	1031
		SET OBJECT SPEED	967
		SET OBJECT TEXTURE	807
		SET OBJECT TO CAMERA ORIENTATION	873
		SET OBJECT TRANSPARENCY	810
		SET OBJECT WIREFRAME	783
		SET PARTICLE CHAOS	1110
		SET PARTICLE EMISSIONS	1108
		SET PARTICLE FLOOR	1112
		SET PARTICLE GRAVITY	1110
		SET PARTICLE LIFE	1113
		SET PARTICLE SPEED	1111
		SET PARTICLE VELOCITY	1109
		SET PIXEL SHADER OFF	1386
		SET PIXEL SHADER ON	1386
		SET PIXEL SHADER TEXTURE	1386
		SET POINT LIGHT	893
		SET RAINBOW SHADING ON	1056
		SET REFLECTION SHADING ON	1057
		SET SHADING OFF	1058
		SET SHADOW POSITION	1051
		SET SHADOW SHADING ON	1045
		SET SHADOW SHADING OFF	1048
		SET SPHERE MAPPING ON	1039
		SET SPOT LIGHT	892

R

RANDOMIZE MATRIX	1214
READ MEMBLOCK	1292
REMOVE LIMB	928
ROLL CAMERA	843
ROLL OBJECT	771
Roman candle	1121
ROTATE CAMERA	838
ROTATE LIGHT	895
ROTATE LIMB	920
ROTATE MATRIX4	1371
ROTATE OBJECT	768
ROTATE PARTICLES	1106
Rotation	748
Absolute	769
Relative	769

S

SAVE MESH	913
SAVE OBJECT	949
SAVE TERRAIN	1193
SCALE LIMB	922
SCALE LIMB TEXTURE	925
SCALE MATRIX4	1370
SCALE OBJECT	775
SCALE OBJECT TEXTURE	802
SCALE VECTOR3	1354
SCROLL LIMB TEXTURE	927
SCROLL OBJECT TEXTURE	808
Seamless tiling	804
SEND NET MESSAGE	1401, 1404
SET ALPHA MAPPING ON	1044
SET AMBIENT LIGHT	888
SET BLEND MAPPING ON	1041
SET BUMP MAPPING ON	1038
SET BSP CAMERA	1173
SET BSP CAMERA COLLISION	1167
SET BSP CAMERA COLLISION RADIUS	1169
SET BSP COLLISION HEIGHT ADJUSTMENT	1170
SET BSP COLLISION OFF	1171
SET BSP COLLISION THRESHOLD	1171
SET BSP MULTITEXTURING	1173
SET BSP OBJECT COLLISION	1167
SET BSP OBJECT COLLISION RADIUS	1169
SET CAMERA ASPECT	846
SET CAMERA FOV	847
SET CAMERA RANGE	848
SET CAMERA ROTATION	840
SET CAMERA TO FOLLOW	853
SET CAMERA TO IMAGE	871
SET CAMERA TO OBJECT ORIENTATION	873
SET CAMERA VIEW	845

SET TERRAIN HEIGHTMAP	1187	V	
SET TERRAIN LIGHT	1190		
SET TERRAIN SCALE	1187		
SET TERRAIN SPLIT	1191	Vertex	749
SET TERRAIN TEXTURE	1188	Vertex buffer	785
SET TERRAIN TILING	1189	Vertex data buffer's structure	1266
SET TEXTURE TRIM	1226	Vertex shader	1376
SET VECTOR3	1352	Video texture	805
SET VECTOR3 TO CAMERA POSITION	875		
SET VECTOR3 TO CAMERA ROTATION	876	W	
SET VECTOR3 TO PARTICLES POSITION	1116		
SET VECTOR3 TO PARTICLES ROTATION	1116		
SET VERTEX SHADER STREAM	1386	WAN	1390
SET VERTEX SHADER STREAMCOUNT	1386	Wireframe	749
SET VERTEX SHADER VECTOR	1386	World	1164
SET VERTEXDATA DIFFUSE	1257	World axes	744
SET VERTEXDATA NORMALS	1254	World units	747
Sky spheres	828	WRITE MEMBLOCK	1284
SET VERTEXDATA POSITION	1250	WRITE MEMBLOCK	1290
SET VERTEXDATA UV	1256		
Shaders	1376	X	
Shadows	1045		
SHIFT MATRIX	1227		
SHOW LIGHT	888		
SHOW LIMB	928	XROTATE CAMERA	840
SHOW OBJECT	776	XROTATE OBJECT	766
SHOW OBJECT BOUNDS	1072		
SHOW PARTICLES	1104	Y	
Skybox	1197		
SNOW PARTICLES	1118		
Solitaire	994	YROTATE CAMERA	841
Spaceship	1122	YROTATE OBJECT	767
Spot lighting	886		
SQUARED LENGTH VECTOR3	1356	Z	
Standard DLLs	1462		
Static collision	1068		
STATIC LINE OF SIGHT	1093		
STATIC LINE OF SIGHT coordinates	1095	ZROTATE CAMERA	841
STOP OBJECT	968	ZROTATE OBJECT	767
String table	1449		
Subdivisions	1212		
SUBTRACT MATRIX4	1368		
SUBTRACT VECTOR3	1357		
Surface reflection	1028		
Switching between cameras	866		

T

TCP/IP	1391
Terrain	1180
TERRAIN POSITION	1183
Textures	750
TEXTURE LIMB	923
TEXTURE OBJECT	798
TEXTURE TERRAIN	1183
Tiling	801
TOTAL OBJECT FRAMES	966
TURN CAMERA	842
TURN OBJECT	770
TRANSLATE MATRIX4	1371
TRANSPOSE MATRIX4	1372

U

UNGLUE OBJECT	934
UNLOCK VERTEXDATA	1250
UPDATE MATRIX	1214

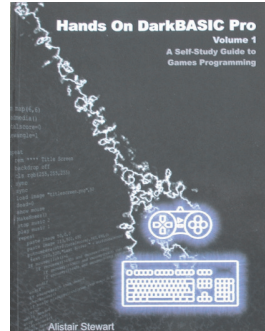
Also Available

from our website www.digital-skills.co.uk

Hands On DarkBASIC Pro Volume 1 742 pages

Contents include:

- | | |
|-----------------------------------|----------------------------|
| Background Concepts | Video Cards and the Screen |
| Starting DarkBASIC Pro | File Handling |
| Data | Handling Music Files |
| Selection | Displaying Video Files |
| Iteration | Accessing the Keyboard |
| Drawing Statements | Mathematical Functions |
| Modular Programming | Images |
| String Functions | Sprites |
| The Game of Hangman | Sound |
| Arrays | 2D Vectors |
| The Game of Bull and Touch | Two-Player Space Duel Game |
| Advanced Data Types and Operators | Using the Mouse |
| Bitmaps | Using a Joystick |



Hands On Milkshape 338 pages (also available as an ebook)

Contents include:

- Background Concepts
- Milkshape Basic Controls
- Principles of 3D Construction
- Vertices, Edges and Faces
- 3D Primitives
- Manipulating Vertices
- Reshaping Meshes
- Extrusion
- Using Milkshapes Additional Tools
- Groups
- Creating Models of Real World Objects
- Texturing Your Model
- Animation
- Exporting Your Model to DBPro

